Technische Universität Berlin

FOG COMPUTING PROJECT REPORT

March 24, 2019

Jacek Janczura, Marina Mursa, Clemens Peters,

Nursultan Shabykeev, Ali Karaki, Marius Schidlack, Alexandre Rozier

Table of Contents

List of Figures

List of Listings

Chapter 1

Introduction

Author: Marina Mursa

In this introductory Chapter the rationale for this study is explained and an overview of the project is provided. It states the motivation behind the research, follows through with the goals that were set to be accomplished by the framework as well as the faced challenges. It finishes by giving an overview of the structure of this report.

1.1 Motivation

In a world hooked on technologies, a connection to the outside world is crucial, even during driving. This is the reason why the majority of car manufacturers try to come up with some technological ecosystem/framework that would connect the car and the user to the world around him.

BMW technology package called *ConnectedDrive* has currently over 8 million cars that connect daily to almost 300 micro services that guarantee not only entertainment but most importantly the security of the users. This services range from Map Updates and Real Time

Traffic Information to Concierge Services. Due to such a diversity of services as well as the constantly growing number of worldwide users, the cars become moving data centers.[1]

For a car to have access to the above mentioned services it requires to make a connection to the BMW servers. Aggregating these connections by time, one type of data that results is represented in the form of requests to connect to BMW servers per second **(requests/sec)**. Each time a service is interacted with, a request is registered, this way, the data is generated continuously. It's worth noting that there exist only a few BMW data centers worldwide that handle this type of information. They are divided in geographical regions like Australia, USA and Europe. The last one includes not just the Continental Europe, but basically the rest of the globe, excluding the first two regions. This indicates how big the amount of handled data is.

The visualisation of such data can be seen in Fig. 1.1.

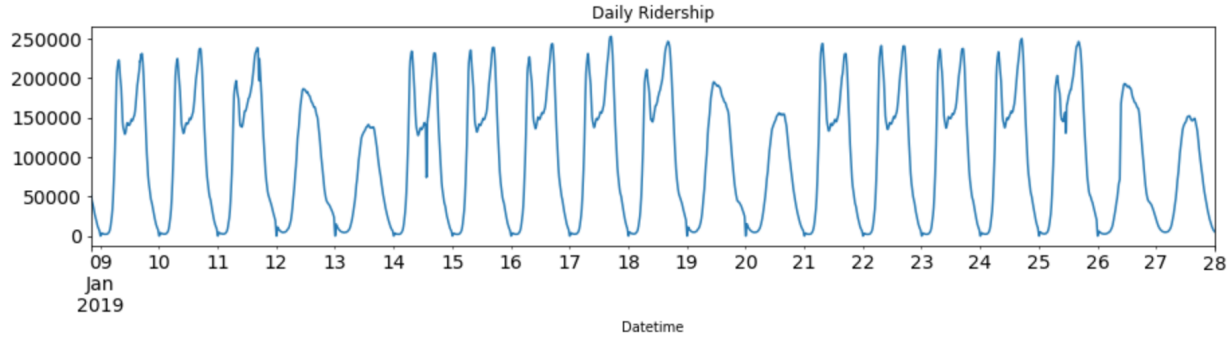


Figure 1.1: Visualisation of BMW data in form of requests per second

A closer inspection of the graph can reveal the following insights:

1. A daily pattern can be observed that corresponds to the intuitive rush-hour peaks. One can notice those elevations around the expected 7:30 and 18:00 (+/- 30 min).

2. A week pattern emerges, where the week days (Monday through Friday) have a strong resemblance, and the weekend days (Saturday and Sunday) have a a slightly changed, more Gaussian pattern.

[1]For more information about ConnectedDrive see https://www.bmwoffreeport.com/blogs/827/what-is-bmw-connecteddrive/

Having such huge amounts of data generated every second, one wonders how can it bring more value to the framework. This way, the challenge of this project was to identify ways to extract that hidden existing knowledge.

1.2 Use Cases

The following section will introduce the two identified use cases for handling the data, the goals that were set for each of them, as well as the challenges each of them pose.

1.2.1 Use Case 1 - Anomaly Detection

Looking back at Fig. 1.1, one can observe a drop in the graph on the 14th of January, that deviates from our expectations. Because the data is a representation of the number of requests to connect to the BMW servers, this drop can indicate either a problem that occurred in processing the received requests or an issue in receiving them, like a power breach.
Being able to identify these deviations that do not correspond to the "normal" distribution of data, would allow The BMW team to react faster in solving the issues behind them. Of course, the faster the anomalies are detected, the quicker the team can intervene, thus assuring that no data is lost due to technical problems, and that the user gets the full ConnectedDrive experience.

Goal

This way, the goal for this use case is to develop a framework, based on AWS service, that would be able to detect in a real-time environment a deviation from the patterns mentioned above, label it as an *anomaly*, and escalate a notification to a tool, e.g. a Slack channel, in the form of an alert accompanied by a small visual representation of the problem.
To achieve this goal the team challenged itself to explore a couple of machine intelligence

solutions that deal with anomaly detection on time series data. After successful implementation, these models should be compared, and based on their advantages and disadvantages one solution should be chosen for the final product.

Challenges

Analyzing the goal and trying to break it down into smaller tasks revealed a couple of challenges that the project would have to deal with prior to starting the actual developing process. These questions are:

1. *How do we define normal?* As it was mentioned above, an anomaly is described as a deviation from the normal expectation. Taking in consideration the fact that the data is not a linear distribution, but has a specific pattern that changes depending on the hour of the day, stating what normal is becomes difficult.

2. *How big of a deviation is an anomaly?* As all distributions, this data has a degree of noisiness to it. Making the framework too sensible will result in a high number of notification, while defining the anomalies too broad will only detect major issues.

The following chapters will offer more details on the issues and the way the project dealt with them both short and long term.

1.2.2 Use Case 2 - Time series Forecasting

Going back to the the Fig. 1.1, one can observe that the number of requests varies highly during the period of a day. This means that the required computational power to process them should follow the same trend. But because the pipeline that processes all the requests is based on AWS services, the scaling of the infrastructure can be done only post-event. This means that for a short period of time the system still uses higher computational power then

required, which translates in higher maintenance costs.
A prognosis of the data distribution would allow the BMW team to adapt the infrastructure based on the predicted needs, thus reducing the expenses with the AWS services in a predictive manner.

Goal

Thus, the project attempts to incorporate a prediction feature, that would deliver short-term prognosis which would allow for predictive scaling of the infrastructure.
As in the previous use case, the team challenges itself with implementing and comparing a series of solutions. Such an approach offers a better perspective of the issues. Comparing the pluses and minuses of a series of machine intelligence models, gives more confidence in recommending a final solution.

Challenges

As one might intuit, this use case comes with a series of challenges that need to be addressed during the implementation. Such challenges include the following questions:

1. *How to handle holidays?* As one can see in the following chapters, holidays are an exception in terms of their pattern, so they do not obey the trends that we discussed above.

2. *How far ahead can we predict?* Although for predictive scaling the system would need just the prognosis for a couple of hours, finding out the maximum prediction period would help determining the limitations of a model.

The way our goals were achieved, as well as the manner in which these issues were handled and their detailed solutions will be presented in the following chapters.

1.3 Outline

This report paper is structured in the following way:

Next chapter is a complex one that starts by explaining the **structure of the system architecture** of the final product of the project. It continues by giving insights into the way the **data is handled**, detailing also on a temporary solution that was explored in the early stages of the project.

As the project has two main story lines that correspond to each use case, the report continues with **detailed explanations of the trial models** for each of them, providing a theoretical explanation, implementation example and results of validating the models against the BMW dataset.

The paper continues with explanations about the means the results of the above mentioned machine intelligence model are reported to a responsible party.

And finally, the chapter finished with details on the **delivery solution**.

The paper continues by describing the way the project was handled from a managerial point of view. And finishes with a list of recommendations.

Chapter 2

System architecture

2.0.1 Introduction

Author: Jacek Janczura

Before diving into the details, we strongly believe that there is a need to highlight the huge challenge that we faced during the early development process.

Our greatest obstacle was the fact that the team had no access to the BMW's streaming data pipeline. The main reasons for that were the fact that this project was in an the early stage of development for BMW, as well as that fact that such a permission would pose a security threat to their team. Later, the BMW team provided only a small piece of an offline, aggregated data. More detailed explanation of how the data was handled will be presented in *section 2.1.1*. The initial lack of data was the reason why the team had to be creative and try to find some work-arounds. We started the project by using some sample data found online and then after getting familiarised with the process of handling the data, we generated a mock of the streaming data, process explained in *section 2.1.4*.
This way, the following section will presents not only the architecture of the final delivered product, but will also include temporary solutions that worked for the first development

stages of the product. We insisted on presenting them to the reader, as they offered important expertise in data handling.

2.0.2 High-level description of the pipeline architecture

Author: Jacek Janczura

In the previous introductory section we talked about the faced "data issue". Due to this challenge, we decided that the best approach towards data would be to handle it in a parallel manner. We chose to divide our pipeline into two main sub-components, one dealing with **streaming data** and the other one with **offline data**.

- **The Offline (or Fixed) approach**. Due to the lack of real time data streaming, we needed to prepare, test and train various machine learning models (DeepAR, RCF and Mean Predictor) for anomaly detection (RCF, Mean Predictor) and prediction (DeepAR, Mean Predictor, Holt-Winter's). At first, we conducted a research and picked the aforementioned models as the most promising ones. Then we deployed those models on AWS SageMaker to test them using faked data from the internet with some anomalies, as well as self generated data in a format of BMW data. As the models worked, we could export them as endpoints ready to be fed with real streaming data.
- **The Streaming data approach**. Streaming data part is a pipeline, which should receive real time data or a mock receiving real time data, preprocess it and feed ML models deployed as an AWS Sage Maker endpoints to detect the anomalies.

Let's explain those two approaches step by step, using the fig. 2.1.

Please note that this figure does not show the Kinesis-based RCF pipeline 2.1.4, which differs significantly from the example presented below (as it was developed independently). For more details regarding this particular pipeline, please read section 2.1.4.

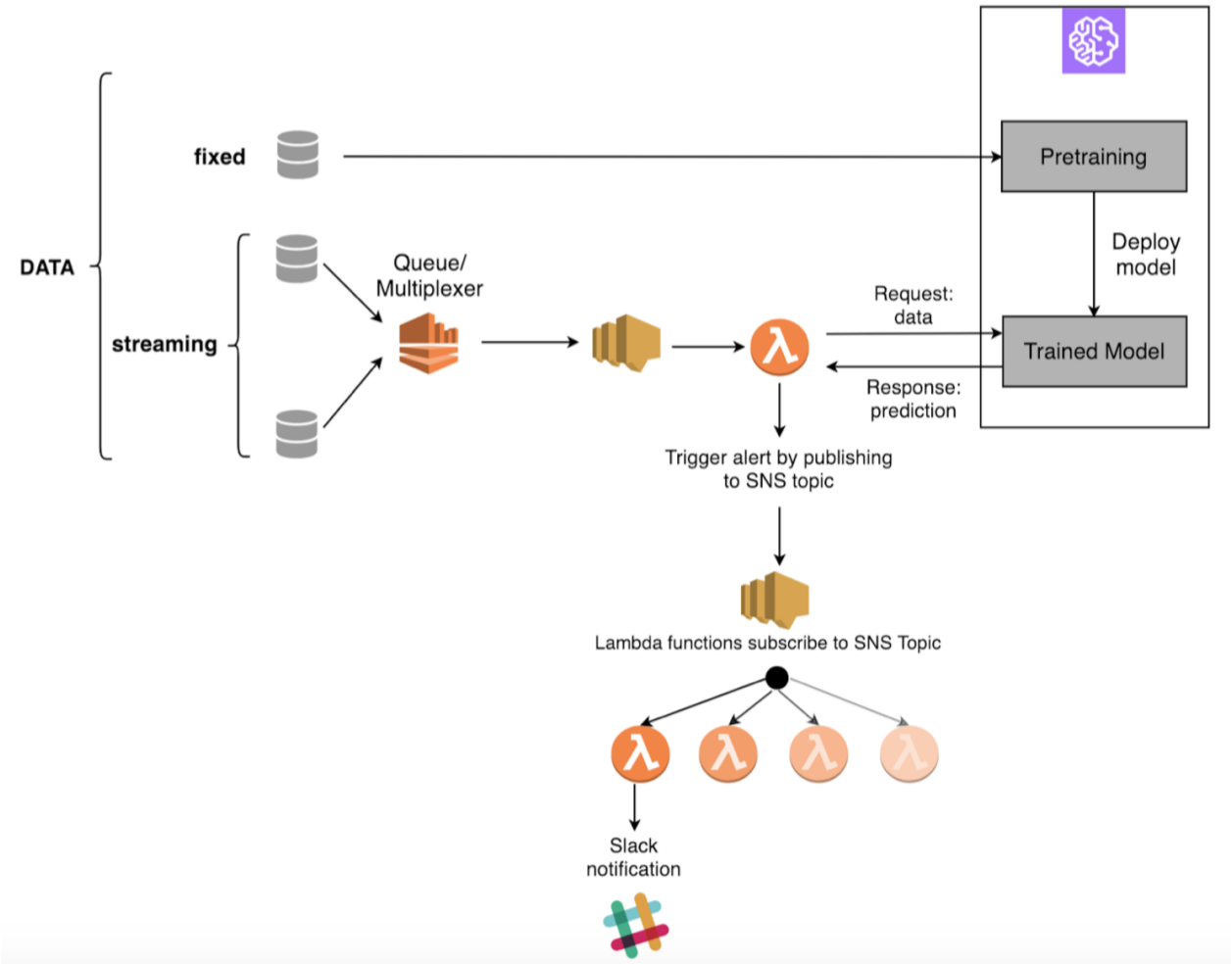


Figure 2.1: High-level system architecture, excluding Kinesis-based RCF pipeline

- **Offline / Fixed**

  1. Offline BMW's data resides into a specific S3 bucket, ready to use. It was provided in two forms: default flowlogs files and preprocessed files (c.f. `mock_data/output.json.example`).

  2. AWS Sagemaker [11] is used to train several machine learning models (DeepAR [6], Mean Predictor [2.2.2] and Random Cut Forests [4]) from the S3 bucket. It is also responsible for exporting them as endpoints, which implies that they are reachable via HTTP requests to run inferences.

- **Streaming**

  As explained in the previous section 2.0.1, we built a system providing us with mock of streaming data that we aimed to process as if it was uploaded by the BMW pipeline.

  1. The streaming data appears regularly in a specific (configurable) S3 bucket.
  2. Each new batch of data arriving to this bucket triggers a new SNS Event [1], which on upload triggers AWS Lambda [3] functions.

     Those functions have diverse roles, but are in most cases used to contact our models exported as SageMaker endpoints. The models are the core elements containing the business logic of the project. They request our exported models for either anomaly detection or forecasting.
  3. Finally, once the results are received, they are forwarded to either the **Chatbot API**, which calls the user to notify them about the anomalies and allows them to make an action, or to our own Slack channel, for real-time monitoring.

2.1 Data processing approaches

2.1.1 Introduction

From the start of the project, our goal was to work on streaming data, piped in a continuous fashion to BMW's processing infrastructure. We encountered several challenges, which explains why we chose to handle data in various ways. Below is the timeline of the different solutions we came up with to deal with each problem:

1. Since our BMW colleagues needed time to hand us their data, we came up with the idea of generating our own Flowlogs (2.1.2,2.25). This allowed us not to loose time, but the generated data lacked the daily patterns of real-life data, so this approach was just temporary.

2. Then, once the real-life data was delivered to us, we could train some models (DeepAR, RCF, MeanPredictor) on it (2.1.3), but were still lacking their streaming version.

3. Indeed, we also wished to build tools able to process the original streaming data, but due to security concerns, we couldn't simply plug ourselves to BMW systems, and had to some extent write some tools mocking these data streams:

    - We wrote a script generating a continuous stream from BMW's real-life data, in order to test the behavior of our whole processing pipeline (2.1.4)
    - We also wanted to test how the RCF algorithm works on data streams, and for this we had to build a Kinesis-based pipeline(2.2.3).

Keeping the evolution of the project in mind is important to understand the choices we made regarding the pipeline architecture. 2.1

2.1.2 Data generation

When it comes to data in Machine Learning models, in most cases the more training data there is, the better results we can achieve. Unfortunately in case of this project, we did not have the desired amount of data to get the appropriate results from the models. As a short term solution for this problem, the team decided to generate some data that has random anomaly spikes. The section below provides a description of data generation pipeline.

Flowlogs stream generation

Author: Nursultan Shabykeev

Prior to getting real data from BMW, we decided to generate the data with random high anomaly peaks. We needed this artificial data to start training and testing our models.
To implement this, we decided to write 2 types of services. The first one continuously generates sequential http-requests to the receiver, hereby, creating 'normal' behaviour of a system. The second one awakes randomly 2 times per day and sends multiple requests within several minutes. This leads to an anomaly situation. Figure 2.2 demonstrates a pattern of our artificially generated data.

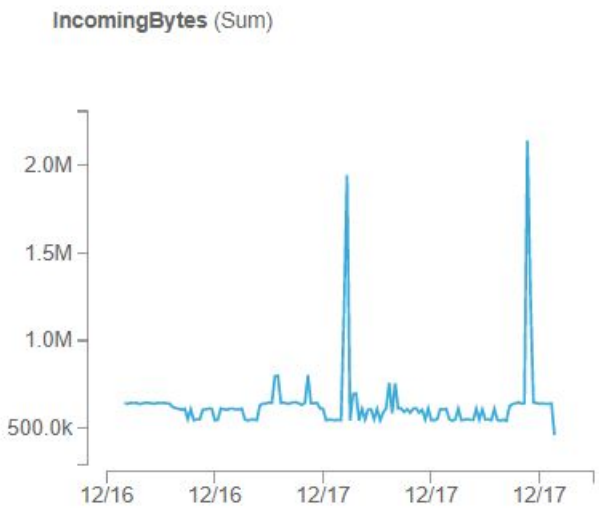


Figure 2.2: The pattern of the generated data

Architecture

After studying services that AWS offers for capturing continuous real-time data, we came to the following architecture. We placed two types of sender micro-services into separate EC2 instances within a common Amazon Virtual Private Cloud (VPC)[1]. Our receiver is basically an EC2 instance within another VPC. In essence, VPC is a virtual network environment that enables a user to easily manage resources in it. One of its useful features is Flow Logs[2]. Flow

[1]https://docs.aws.amazon.com/vpc/index.html
[2]https://docs.aws.amazon.com/vpc/latest/userguide/flow-logs.html

Logs allows us to access information about all inbound and outbound traffic to and from instances in VPC. This information is represented in records. A record consists of fields that describe the traffic flow and has the following format:

```
<version> <account-id> <interface-id> <srcaddr> <dstaddr> <srcport> <dstport> <protocol> <packets> <bytes> <start> <end> <action> <log-status>
```

Figure 2.3: Flow Log record format

In our case, we enabled Flow Logs for a receiver's VPC to capture requests from senders. We did not track responses from the receiver. Figure 2.4 shows a general pipeline for data generation.

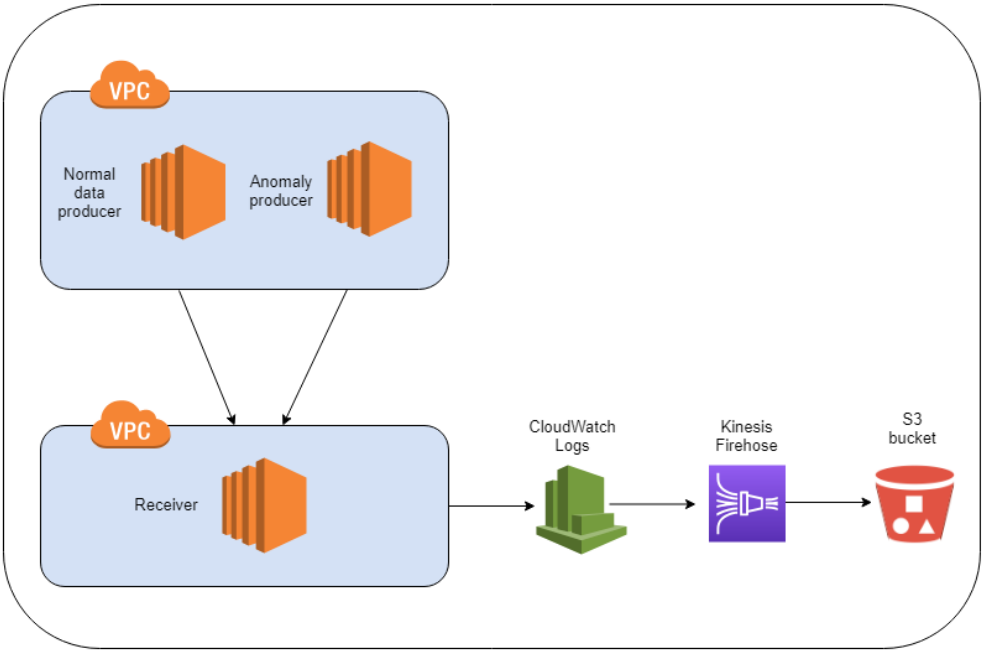


Figure 2.4: Data generation pipeline

Working with streaming data

VPC Flow Logs can be published either to Amazon CloudWatch Logs or directly to Amazon S3. We decided to publish to CloudWatch Logs because later we would like to try out Kinesis

real-time data analysis services from AWS on the streaming log data. Amazon Kinesis Data Streams (KDS) is a set of services provided by AWS to work with real-time streaming data:

> Amazon KDS is a massively scalable and durable real-time data streaming service. KDS can continuously capture gigabytes of data per second from hundreds of thousands of sources. . . . The data collected is available in milliseconds to enable real-time analytics use cases. . . .[2]

From this set of services, we used Kinesis Firehose to load VPC Flow Logs. Creation of a Firehose delivery stream can be done as follows, replacing the placeholder values for RoleARN and BucketARN with the role and S3 bucket ARN where flog logs will be stored:

```
aws firehose create-delivery-stream \
   --delivery-stream-name 'my-delivery-stream' \
   --s3-destination-configuration \
  '{"RoleARN": "arn:aws:iam::123456789012:role/FirehosetoS3Role", "BucketARN": "arn:aws:s3:::my-bucket"}'
```

Figure 2.5: Creation of Kinesis Data Firehose delivery stream

After the Firehose delivery stream is ready and in an active state, we direct the streaming data to it from CloudWatch Logs. This can be done through a subscription filter[3]. The subscription filter connects a Firehose stream and CloudWatch logs and starts a flow of real-time log data to a Firehose delivery stream.

```
aws logs put-subscription-filter \
    --log-group-name "CloudTrail" \
    --filter-name "Destination" \
    --filter-pattern "{$.userIdentity.type = Root}" \
    --destination-arn "arn:aws:firehose:region:123456789012:deliverystream/my-delivery-stream" \
    --role-arn "arn:aws:iam::123456789012:role/CWLtoKinesisFirehoseRole"
```

Figure 2.6: Creation of subscription filter

After the subscription filter is set up, CloudWatch Logs sends all the incoming log events that match the filter pattern to Amazon Kinesis Data Firehose delivery stream. It might

[3]https://docs.aws.amazon.com/AmazonCloudWatch/latest/logs/SubscriptionFilters.html

take a few minutes before the data will be saved into S3. However, it is possible to monitor a size of incoming data and other metrics in a *Monitoring* tab in a Firehose console page.
While creating the subscription filter, it is important to define properly IAM roles and permission policies and correctly associate the permissions policy with the role. The full description of establishing the connection between Kinesis streams and CloudWatch Logs is available in an official documentation [10].
As stated above, we used Kinesis services to get introduced to real-time data analysis tools available in AWS. Later, we installed Kinesis Data Analytics[4] as a before step before sending the data to S3 bucket (see figure 2.4). This way we tried to make a simple analysis and, for the first time, apply Random Cut Forest algorithm to the artificially streamed data. Even though, the work described in the current section was a temporary implementation, it generally helped us to come up later with final solutions.
As the real data was eventually published in an S3 bucket, we did not need the data generation pipeline anymore. That is why we currently do not deliver it, only the code for generating requests is available in GitHub. The version that is implemented in Java is available in: `utils/flowlogs-stream-generation/`.
Another approach that is implemented in Python is described in the following section.

Data Generator - Python

Author: Ali Karaki

The Dummy Data Generator in Python variant is an alternative version of the one being discribed above. It simulates a stream of https requests to a specific IP (EC2 instance inside a VPC). It is highly configurable from scheduled time for requests, has a dynamic server IP for the EC2 instance, and a placeholder for the number of desired requests being sent. We needed an easy to customize script that can generate data fast. This script fulfills this requirement. It had no complex dependencies and has a pretty fast executions.

[4]https://aws.amazon.com/kinesis/data-analytics

In concept, the workflow to generate dummy data was to do the following:

1. It sends HTTPS requests with our scripts to an EC2 instance.
2. The logs from EC2 instance are saved in CloudWatch.
3. VPC Flowlogs are saved inside a bucket on S3.
4. Flowlogs aggregator lambda function is used to aggregate all the files into actual data set.

A request may be rejected or accepted when it reached the EC2, by comparing it to the IP of a machine from the VPC (it can be seen in Fig. 2.7).
Requests are sent to a specific IP, the requests are categorized into 2 sections: successful and failed with codes 200 and 599 respectively.
The sent data, that is already separated into successful and failed requests, and the number of requests are the variables used for anomaly detection.
Anomalies detected with this data correspond to the number of requests recorded in a specific timestamp. So the code logic goes as following:

1. Send failed requests and successful requests to the EC2 instance
2. Randomly send those requests in 2 different methods
3. Every day from Monday to Sunday, randomly send only 1 very abnormal number of requests and 1 very high number of requests

As this script is used only to generate the data the script runs in an infinite loop until it is disabled manually, with a possibility to configure the number of requests and date of sending them.

```
#Def for a request, stops when the looper is 0
def handle_request(Req):
    #Failed requests for testing
    if (Req.code == 599):
        print('\n',Fore.RED + 'Request Code, FAIL = ' ,Req.code, '\n')
        print(Style.RESET_ALL)
    else:
    #Successful requests for testing
        if (Req.code == 200):
            print('\n',Fore.GREEN + 'Request Code, SUCCESS = ' ,Req.code, '\n')
            print(Style.RESET_ALL)

    print(type(Req), '\n')
    # If you want to print headers, uncomment this
    #print('Request Headers = ',Req.headers)
    global i
    i -= 1
    if i == 0:
        ioloop.IOLoop.instance().stop()

#Requests Looper, you can change the IP of the server down below
def Random_Requester(Number_Of_Requests):
    http_client = httpclient.AsyncHTTPClient()
    global i
    for x in range(Number_Of_Requests):
        i += 1
    #Send data to this IP
        http_client.fetch("http://54.147.181.67", handle_request, method='HEAD')
    ioloop.IOLoop.instance().start()
```

Figure 2.7: Data generation in a python script

2.1.3 Fixed data approach

Author: Marius Schidlack

After working with our fake data streams for some time, we were provided with some real data by BMW. The data was presented in two different formats: Flowlogs and Metrics. They contained network traffic information starting from December 21, 2018. Flowlogs contain raw information about network traffic, e.g. which IP-adress made a request to which server at what time along with other details.

In this section, we will focus on how we used the Metrics files. The Metrics already accumulate

the raw network traffic data from the Flowlogs and only contain information about the number of requests. More specifically, they contain the number of successful requests (with HTTP-Code 200) and failed requests (HTTP-Code 4xx and 5xx) at a certain time. Although the files were named *output-mass.json*, they were not correctly encoded in JSON format. An example entry from one of the files can be seen in 2.1.

```
1  {
2      [...]
3      'response-code-4xx': {
4      [...]
5          'Datapoints': [
6              {'Timestamp': datetime.datetime(2018, 12, 15, 11, 20,
                   tzinfo=tzlocal()),
7              'SampleCount': 6.0,
8              'Unit': 'None'
9              }
10         ]
11     }
12 }
```

Listing 2.1: Example entry from metrics data

For this reason (invalid JSON format) we had to write a custom python script to parse the data (i.e. to read the data from the files and load them into python objects). Once all the files are parsed, we take the *SampleCount* values for each entry, and sort them into buckets according to their *Timestamp*. For the time being, we did not differentiate between successful or failed requests, however, this might be a useful thing to do in the future. The buckets were chosen to be of 5 minute granularity in the final product, however we also experimented with 1 minute buckets. A smaller bucket size means higher responsiveness in a real time anomaly detection model, since in order to react to incoming data, we have to wait until the current bucket is full. On the other hand, if the bucket size gets too small, the machine learning model might get vulnerable to noise in the data. Larger buckets means less number of buckets and compress the training data which allows to train the model faster. Furthermore this can make the model more robust to noisy data.

Next, we summed up the values in every bucket to obtain the raw time series, which can be seen in figure 2.8 in the first graph.

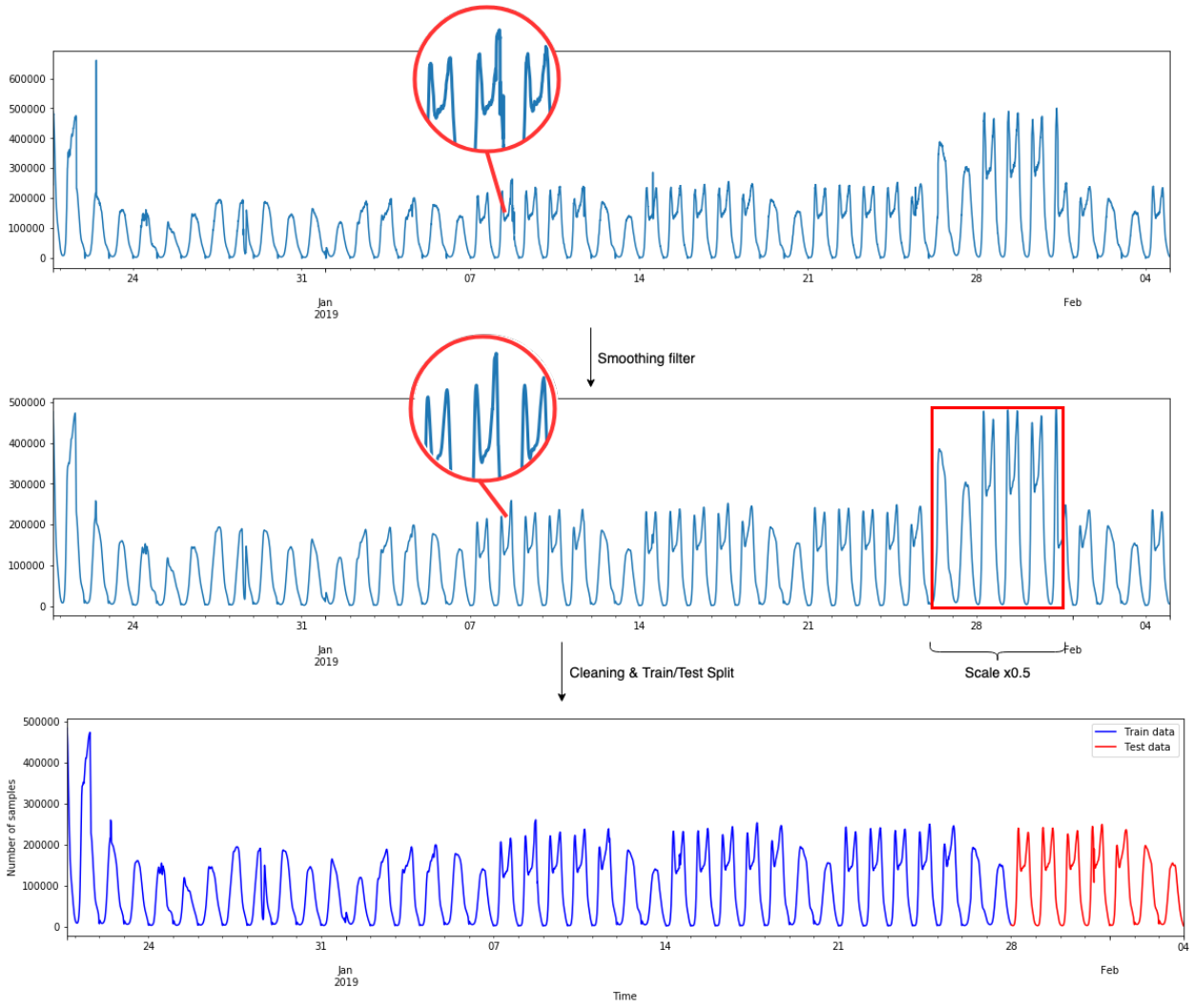


Figure 2.8: Data preprocessing pipeline

Preprocessing and Cleaning

We noticed that the data is still very noisy and contains some impurities. For example, the data between the 26th and the 31st of January was scaled up by a factor of two for unknown reasons (see red box in the second graph in figure 2.8). We reduced the noise by applying a smoothing convolution to the data, and scaled down the aforementioned scaled up time interval (cf. fig. 2.8). One might argue that in time series data, the value of one point in time is highly correlated to the previous ones. Smoothing the time series exploits this

property to get rid of sudden and unexpected jumps in the training data, or in other words, we reduce outliers and make training of our machine learning models more robust. Finally, we separated one week of data and defined it as out test set.

Exporting for Machine Learning Models

For further use with the Machine Learning models described later, the data had to exported in different formats. For the Random Cut Forest and the Mean Predictor, the data can be provided in a simple CSV format with *Timestamp* and *Value*. The DeepAR Model however requires a more complex format which is described in more detail in chapter 40.

Advantages and Disadvantages

While extracting the data from the Metrics files is convenient and simple, this approach has drawbacks: we have no control over how the Metrics files are accumulated. In order to obtain full control over the data stream, one must work with the raw Flowlogs files. An approach that utilizes the Flowlogs data from scratch is described in section 2.2.3.

2.1.4 Streaming data approach

Author: Alexandre Rozier This approach uses a custom script hosted on EC2 . **Code relative to this section can be found in** `utils/streaming-data-mock.py`.
Since we were just provided with a fixed dataset and did not have access to the real BMW data stream, we had to create our own mock of it, to test our solution with real-time data. Its principle is quite simple: we extract one default week from the fixed dataset provided by BMW, and use a script to send over and over this particular week on our AWS infrastructure.

An EC2 instance is spun up via Terraform (2.6.4), then provisioned thanks to an Ansible

script (installation of the correct python version, required dependencies, upload of the source code and execution).

To be more specific, the script triggers a job every 15 minutes, charged with uploading a new batch of data to a specific entry point on AWS S3. Please note that the data is sent synchronously with the current time, that is, at 8am we will send data concerning the timespan 7:45-8:00am from the default week.

We chose to implement this fake stream in order to test the whole pipeline, because it was the closest we could imitate BMW's infrastructure putting new batches of data to our entry S3 bucket. This was useful during the final presentation (where we could show our pipeline processing live data), and also to work as close a possible to a real-life scenario.

2.2 Anomaly detection algorithms

As laid out in the introduction (cf. 1.2.1), we faced the problem of anomaly detection in time series data. In the pursue of solving this problem, we tried out several algorithms of varying complexity, each with their advantages and disadvantages. First, we introduce the Random Cut Forest algorithm in the following section (2.2.1). Subsequently, we put forward a very simple algorithm that we call the Mean Predictor (2.2.2). Finally, we will explain a radically different variant of the Random Cut Forest approach, which incorporates real time data streams into the learning and detection process (2.2.3).

2.2.1 Amazon SageMaker Random Cut Forest

Author: Nursultan Shabykeev

Motivation

Amazon SageMaker RCF is an algorithm for detecting anomalous data points within a data set. Generally, anomalies differ significantly from the majority of the data and usually manifest as unexpected breaks in periodicity or sudden high peaks in time series. After we plotted our data, we could easily see that it contained high spikes and we wanted to know whether the RCF model can identify these data points as anomalies and how well it is able to detect local anomalies. Besides, the fact that Random Cut Forest[5] is a built-in algorithm for Amazon SageMaker and can be easily deployed, additionally motivated us to try it out on our data set.

General idea

The main idea behind the algorithm is that it firstly takes a random sample of the training data and cuts them to the same number of points, i.e., the sample is partitioned into a number of equal-sized subsets. Then, each subset is assigned to an individual tree what leads to creation of a forest. The forest is later used to define whether a new data point is an anomaly. It is done by inserting a new point into the trees and determining how much that changes the complexity of the forest. If an average change of depth and width of trees highly increases, than a data point might be considered as anomalous. Generally, the algorithm assigns an anomaly score for each input data point. The higher a score, the more a data point differs from the majority of the data.

Below, we describe creation and deployment of the RCF SageMaker model and as well as the final results. For the full and detailed explanation of RCF algorithm please see an official documentation [5].

[5]Amazon RCF SageMaker is an implementation for batch data processing, not to be confused with Kinesis RCF which is designed for near real-time data processing.

Training

The data consists of a number of requests (VPC Flowlogs from BMW) aggregated into one minute buckets. To train the model, we used the data over the course of one week that has "normal" behavior. Then, we fed the whole data into the trained model for testing. We will present the test results later. Here, we describe important parts of the training process.

Hyperparameters

RCF model can be tuned with the following hyperparameters:

- *num_samples_per_tree* indicates the expected density of anomalies within the data. As a general rule, $1/num\_samples\_per\_tree$ should approximate the estimated ratio of anomalies to normal points in the dataset. For example, if this value equals to 512, then we expect our data to contain anomalies 1/512 or approximately 0.2% of the time.
- *num_trees* - the number of trees to create in the forest. Each tree learns a separate model from different samples of data. The full forest model uses the mean predicted anomaly score from each constituent tree [5].
- *feature_dim* - the dimension of each data point.

Along with these RCF model hyperparameters, we provide additional parameters like the EC2 instance type on which training will run, the S3 bucket containing the data, and the AWS access role (see figure 2.9). Note that,

- Recommended instance type: ml.m4, ml.c4, or ml.c5
- Current limitation: The RCF algorithm does not take advantage of GPU hardware [5].

```
from sagemaker import RandomCutForest

session = sagemaker.Session()

# specify general training job information
rcf = RandomCutForest(role=execution_role,
                      train_instance_count=1,
                      train_instance_type='ml.m4.xlarge',
                      data_location='s3://{}/{}/'.format(bucket, prefix),
                      output_path='s3://{}/{}/output'.format(bucket, prefix),
                      num_samples_per_tree=512,
                      num_trees=100)

# automatically upload the training data to S3 and run the training job
rcf.fit(rcf.record_set(num_val_array))
```

Figure 2.9: RCF Model training

Deployment

After the model is trained, we can to deploy it on an endpoint. We use `SageMaker Python SDK deploy()` function from the job to create an inference endpoint. The function has two input parameters: the instance type and an initial number of instances. Now using the deployed model, we can perform inference on the data. There are two ways to invoke the trained model:

- Right after training in the same notebook, calling a job's *predict(Test_data)* function
- From elsewhere(e.g., inside a lambda function), using a function *runtime.invoke_endpoint (EndpointName, ContentType, Body)*

Computing an anomaly threshold

RCF algorithm outputs an anomaly score for each input data point. 'Normal' data points are assigned low score values, whereas anomalous data points get high score values. Basically, a data point is considered to be anomalous if it is beyond a specific threshold. It is suggested in a RCF documentation [5], to compute a value of an anomaly threshold as 3 standard deviations from the mean score. Of course, the way to compute the threshold value is subject to change. Furthermore, another question is how to properly compute the threshold in terms

of time, i.e., how long should a threshold stay valid. Depending on business requirements, one can calculate it either once for a whole period of data or for shorter periods of time, e.g., recompute threshold every day.

For our data, we tried different ways of computing a threshold. Firstly, we computed it for the whole period of data, i.e., for almost one month. In this case, just evident peaks as on 2019-01-25 were beyond the threshold. Later, we decided to calculate the threshold per each day and these results are presented in figure 2.10.

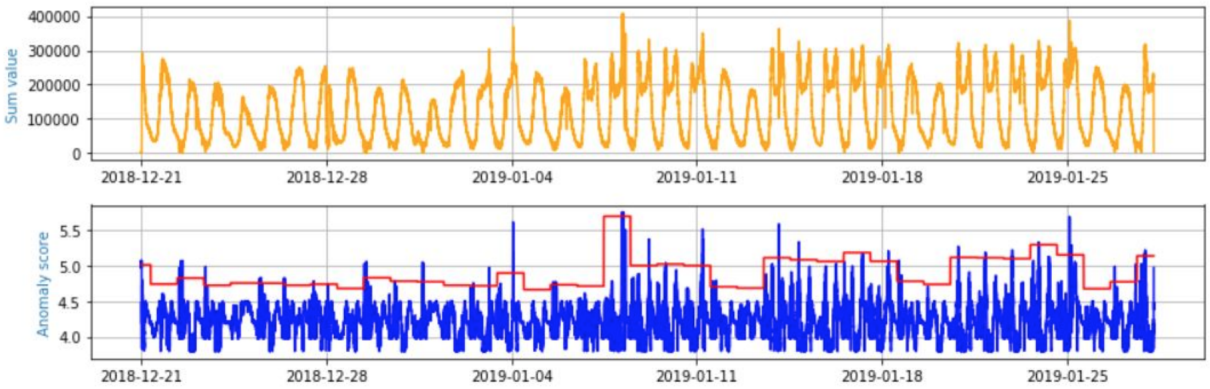


Figure 2.10: Plotted data, anomaly scores, and anomaly threshold values

Results

In figure 2.10, the first plot represents the data, the second one demonstrates test results. The red plot depicts threshold values that we compute for each day. As we can see, not only high peaks are detected as anomalies, but also local downfalls (as on 2019-01-18) and congestion on bottoms (as on 2018-12-29) are anomaly candidates.

In general, SageMaker RCF algorithm detects various types of anomalies on our data. Of course, some of found anomalies might not be anomalies at all in terms of application. One possible solution to this problem could be training the algorithm with more amount of data. In our case, we had the data just over the course of one month. Training a model with

regularly repeating complex patterns might improve overall performance. Another technique is proper adjusting of a threshold, e.g., increasing it on holidays based on previous experience.

2.2.2 Mean Predictor Algorithm

Author: Marius Schidlack

Motivation While sophisticated algorithms like the random cut forest seem to be promising for fitting large amounts of data with complex patterns, they did appear overly complicated for the amount and nature of the data at hand. The data shows a clear weekly periodicity, where the data from one week can hardly be distinguished from another. The only exception is the one week of holidays. In fact, the weeks are so similar, that we can predict the same pattern every week with high confidence. This was the motivation for us to implement a very simple, yet effective machine learning model called the Mean Predictor.

Idea The main idea is already contained in the name Mean Predictor. The objective of the Mean Predictor is to calculate the expected week and standard deviation for each point in a week. Accordingly, we can identify outliers by simply checking if the data point lies within the allowed deviation of the expected value.

Unfortunately, AWS Sagemaker does not offer a Mean Predictor as an Out-Of-The-Box algorithm like the Random Cut Forest. Thus, we implemented it ourselves.

Implementation For calculating the expected week, we sorted every data point into a bucket that is determined by its *Timestamp*. Each bucket has a hash or index associated with them. After distributing all the data into buckets, we calculate the mean of all the values in one bucket, as well as a measure of standard deviation. This process is visualized

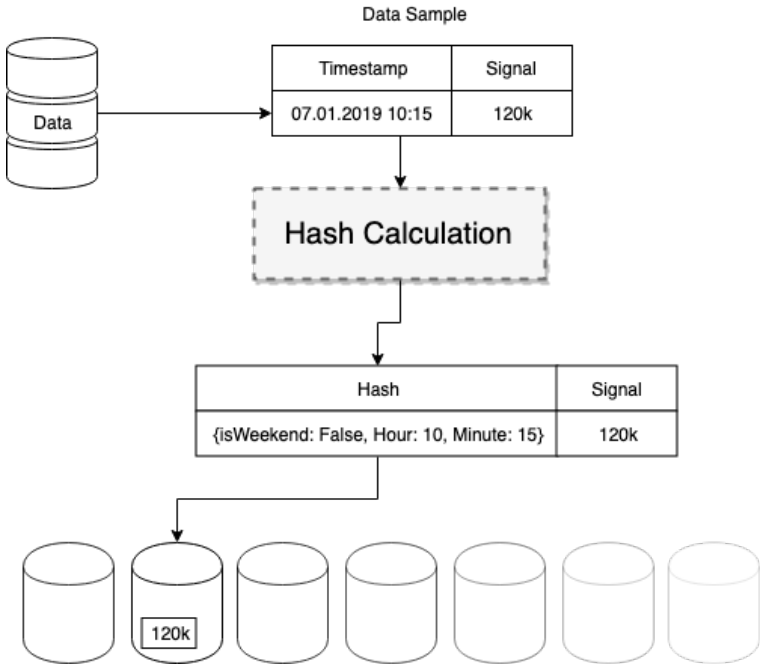


Figure 2.11: Hashing process of Mean Predictor Model

in figure 2.11.

The hashing function is central to this approach, as we can arbitrarily craft the features that we want our mean predictor to represent. Apart from the weekly periodicity of the data, we also noticed a very predictable daily pattern throughout the working week. Hence we decided to put all working days into one set of buckets and all weekend days into a different set. We did this to ensure a reasonable amount of samples for each bucket. The hashing function currently employed in the mean predictor is shown in code example 2.2. As more and more data comes in, the function can be adjusted if necessary.

```
1  def __time_hash(self,t):
2          return (t.weekday() < 5,t.hour,t.minute)
```

Listing 2.2: Time stamp hashing function used for Mean Predictor

As mentioned before, we also require a measure of expectation deviation for determining outliers. The first thing that comes to mind is the standard deviation:

$$\sigma_t = \sqrt{\frac{1}{n}\sum_{i=1}^{N}(s_t^{(i)} - \mu_t)^2} \tag{2.1}$$

Where μ_t denotes the mean value of bucket t and $s_t^{(i)}$ the i-th value in this bucket. However, since we square the distance from the mean, the standard deviation weighs outliers much heavier than conform data, which might not be the desired behaviour for the task at hand, because outliers are what we want to detect. Instead one might want to use the average euclidean distance from the mean:

$$\sigma_t = \frac{1}{n}\sum_{i=1}^{N}\sqrt{(s_t^{(i)} - \mu_t)^2} \tag{2.2}$$

In our case, the choice of deviation measure had little influence on the final results, nevertheless if future training data is noisy and contains a lot of outliers, this might be an option to consider.

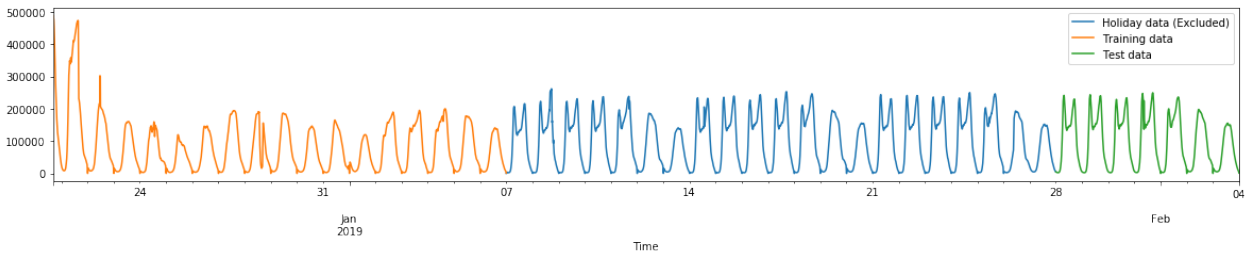


Figure 2.12: Train-Test split of data used for the mean predictor

Training and Prediction Due to the fact that the Christmas holidays are included in the data, a lot of the provided training data is anomalous and therefore not useful for training a model that is supposed to detect anomalies. Accordingly, we excluded this data from training, leaving us with only three weeks of training data. Alternatively one could also alter the hashing function to give holidays a separate set of buckets. Our train-test split is shown in figure 2.12.

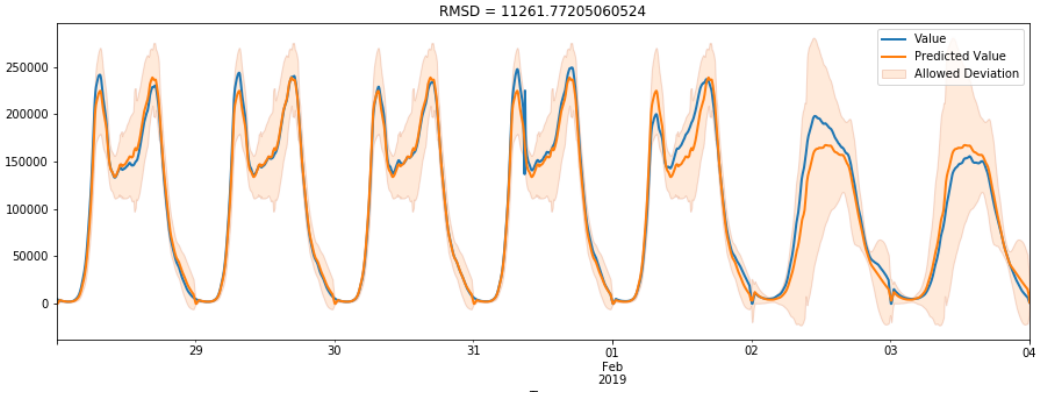


Figure 2.13: One week prediction of Mean Predictor model

Once we have expected value and deviation for each bucket, making predictions is trivial. The model simply takes a time interval and calculates the hash for each timestamp within that interval. We obtain the prediction by simply looking up the expected value and deviation in our table. Figure 2.13 depicts a prediction for the week starting at the 28th of January 2019 (test set) and compares it to the ground truth. For this test data, the prediction is highly accurate and the true values never leave the confidence interval. The RMSE *(Root Mean Squared Error)* of the prediction lies around 11,261. Further investigation is needed to accurately assess the outlier detection performance, since we were lacking test data that contains real outliers. For the time being, by looking at the confidence interval, we can see that sudden spikes or lows in the timeseries could easily be identified as anomalies.

Deployment as Sagemaker Endpoint In order to use our model in the Cloud, we deployed it as an AWS Sagemaker Endpoint. In the following we describe the technical details related to this task. The endpoint was implemented along the lines of a tutorial provided by AWS [6].

Sagemaker Models are deployed from docker images. Hence, we defined a docker file that encapsulates the model into a container. The container image is then built and uploaded to AWS *Elastic Container Registry* (ECR) with the *build_and_push.sh* script. The *train_deploy.py* script provides a convenient command line interface for spinning up a Sagemaker endpoint using the previously uploaded container image. This script can also be embedded into a terraform pipeline. The model requires only one hyperparameter: The granularity or *frequency* with which it should make the timeseries predictions. Furthermore it needs a path to the training data located in a S3 Bucket.

At its core, the Sagemaker endpoint is nothing but a REST-API that takes training jobs and prediction requests and forwards it to the Mean Predictor. Once deployed, the endpoint takes prediction requests in the following JSON format:

```
1 {
2       "start": "YYYY-MM-DD HH:MM:00",
3       "end": YYYY-MM-DD HH:MM:00
4 }
```

It will respond with a time series prediction within the *start-end* interval, depending on the aforementioned *frequency*. The response is delivered in the following CSV-Format:

Timestamp	Value	Std
2019-01-05 10:15:00	100	7
2019-01-05 10:20:00	96	13
2019-01-05 10:25:00	112	8
...	...	...

[6] https://github.com/awslabs/amazon-sagemaker-examples/blob/master/advanced_functionality/scikit_bring_your_own/scikit_bring_your_own.ipynb

Similar to the Random Cut Forest, the model will not do outlier detection by itself, but rather provide the user with the necessary information to do so. In order to detect an outlier, one must simply check if a specific data point lies within the bounds of allowed deviation at that time. The user can arbitrarily scale the allowed deviation to their requirements. Such functionality is easily implemented within an AWS Lambda function, which allows seamless integration of the model into the Cloud Architecture. Furthermore, it allows to use the same model for anomaly detection and prediction at the same time.

Conclusion The Mean Predictor is of course a very minimalistic approach: in its current state, we can only incorporate a single source of information, it is not able to leverage additional data, e.g. weather data to improve its prediction. The model is not able to learn features of the time series by itself. As the prediction will resemble a regular working week every time, the model is of little use during holiday periods, however, once enough data is gathered, the hashing function can be manually adjusted to differentiate between holiday and working weeks. With the small amounts of training data and the selected test set, the prediction of the Mean Predictor vastly outperforms the other, more complex, Machine Learning models that we examined with an RMSE of just 11,261.

2.2.3 Real time anomaly detection with Kinesis

Author: Clemens Peters

Motivation

This approach is inspired by a Medium blog post[9]. Note that this approach is entirely different to the aforementioned ones, and does **not** build on the architecture shown in figure 2.1. Instead, a different architecture is incorporated which will also be described in this

section. The goal of the approach is to move the anomaly detection closer to a real-time setup. This means to continuously process the most recent data instead of analyzing data from the previous month or year to find anomalies. To give some more motivation about why this is actually an important issue, here is a quote from a job description from Netflix[8]:

> "Netflix Operational Insight Team is the team responsible for building common infrastructure to collect, transport, aggregate, process and visualize operational metrics. We build powerful systems to allow everyone at Netflix visibility into the state of our environment at both a macro and micro level."

The fact that a leading and modern company like Netflix builds a whole team just dedicated to checking and visualizing the current system state in real time, underlines how important it is to know at every moment how your environment performs.
AWS Kinesis is a very suitable tool for this problem for multiple reasons. First of all, Kinesis is designed to process streaming data. This means we can ingest real-time data which is a perfect fit for our given VPC Flowlog data. Second Kinesis comes with Kinesis Data Analytics already built in. Amazon Kinesis Data Analytics offers easy ways to analyze streaming data, gain actionable insights, and respond to business needs in real time. In addition to that there is a SQL function called "Random cut forest with explanation" which is described by AWS as follows:

> Computes an anomaly score and explains it for each record in your data stream. The anomaly score for a record indicates how different it is from the trends that have recently been observed for your stream. The function also returns an attribution score for each column in a record, based on how anomalous the data in that column is. For each record, the sum of the attribution scores of all columns is equal to the anomaly score. [4]

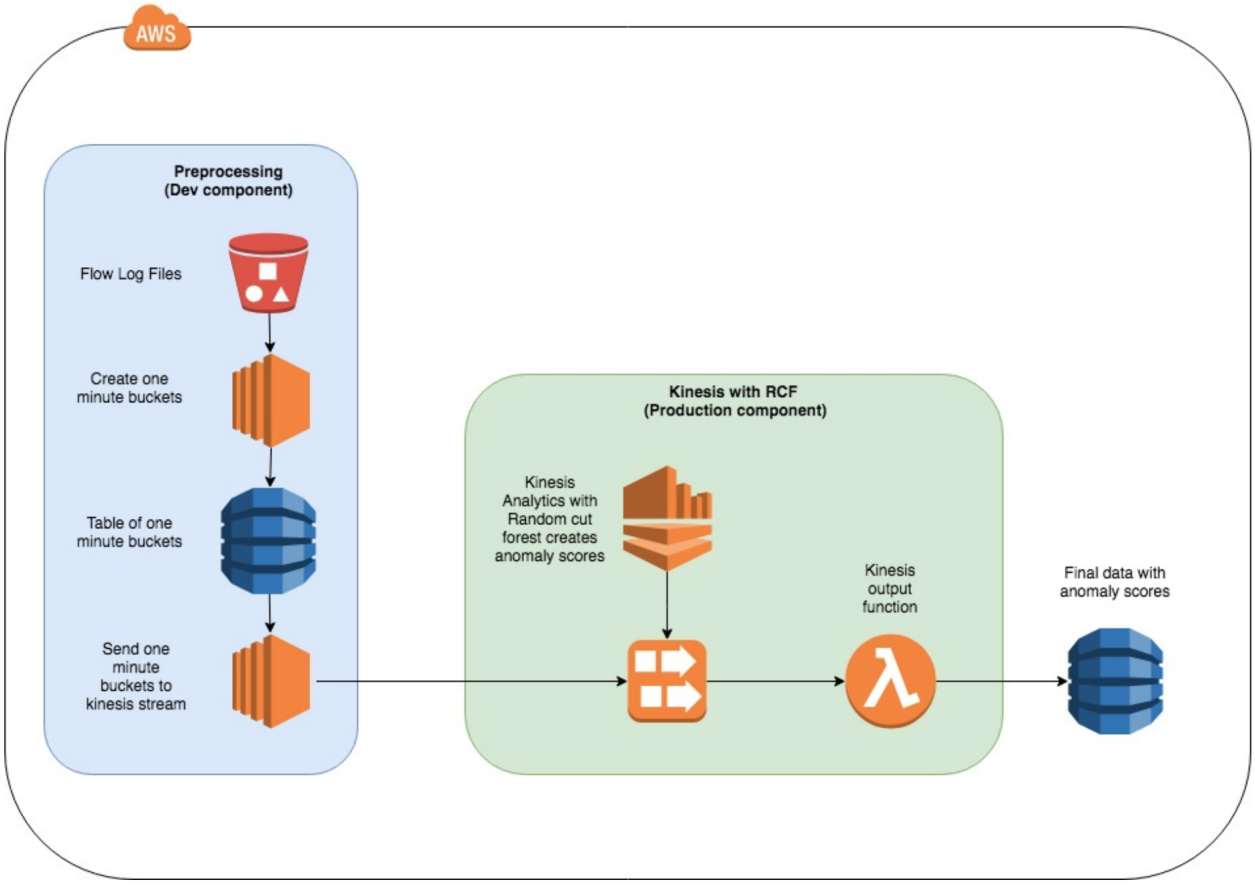


Figure 2.14: Medium Kinesis Random Cut Forest Setup

Architecture

Given this prior knowledge, we will now dive further into the system setup. First let us get an overview of the system as it was implemented during the project phase:

To start with, it should be noted that all the preprocessing components on the left (blue area in figure 2.14) would not be part of a production system. In a production system we would have a continuous stream of VPC Flowlogs which needs to be handled and preprocessed, whereas in the project setup we had all the Flowlogs data from the past in a S3 bucket.

Preprocessing components (blue area in figure 2.14)

In the following section we will look at the preprocessing setup in detail, keeping in mind that this is not created to run in production.

S3: Flowlogs Files

All the Flowlogs data provided by BMW lies in an S3 bucket called *fog-bigdata-bmw-data*. As mentioned in section 17, there are two types of data: first the Metrics, which already contain summaries about how many requests were made to the system during a certain time interval. Another issue here was, that the time intervals are of different size (sometimes five minutes, sometimes four or six or something similar). Of course this is not a good basis to run our anomaly detection, because we cannot know what value we expect (since the size of the intervals are different). Therefore the second type of data is more suitable, namely the raw VPC Flowlogs. However, we want to look at how many requests are made to our system per minute and thus we need to do some preprocessing.

EC2: create one minute buckets

At this point, the first script with the name *createOneMinuteBuckets.py* comes in. This script is executed on an EC2 instance and reads all the flow log records from the S3 bucket. It creates one-minute buckets from the Flowlogs, so that we know for every one minute time interval how many requests were made to the system.

DynamoDB: table of one minute buckets

The information about these one minute buckets is then written into a DynamoDB table (called `medium_bmw_data_to_kinesis`). For the sake of this project the data from the Kinesis table was then exported to a CSV file and the information about the weekdays was added (currently done in Excel). This means that for every one minute bucket we know the hour and the minute and also on which weekday it was recorded. This might be relevant because the expected number of request at 8am on a Sunday can be very different than 8am on a Monday, especially because the requests come from driving cars. Of course this step will not

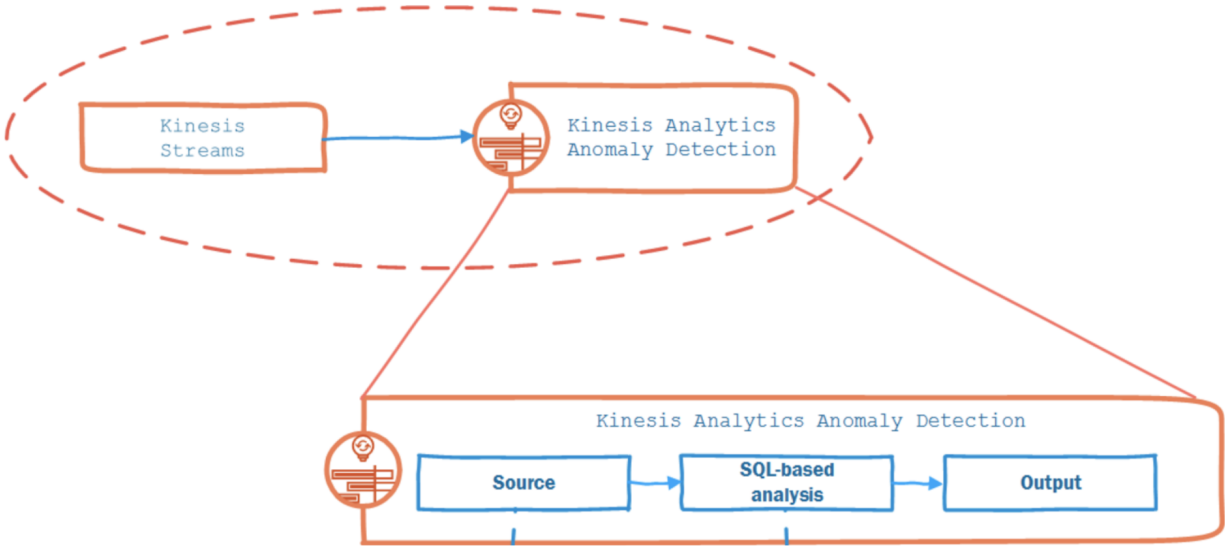
be part of a production system (as mentioned before).

EC2: send one minute buckets to Kinesis stream

The next script is called *generateAnomalyScores.py*. The script and the final CSV file is uploaded to an EC2 instance again. The script takes the data from this CSV file and sends it to the Kinesis stream called *medium_VPCFlowLogs*. An alternative setup could be to read the data directly from DynamoDB and add the information about the weekday in the python script (instead of using CSV and Excel).
This is where the preprocessing part (which would look different in a production system) ends and where the production component (green area in figure 2.14) comes into play.

Production component (green area in figure 2.14)

In this section we assume that we already have all the data we need in the right format. This is the part of the system where the anomaly detection happens. This part of the setup can be used in a production environment in the same or in a similar way. All the preprocessing is done and the data is already fed into the our Kinesis stream.



Kinesis Data Analytics with RCF

Source: https://medium.com/@devfire/real-time-anomaly-detection-in-vpc-flow-logs-part-5-anomaly-detection-d1fc9b61baf8

The Kinesis stream has a Kinesis Data Analytics application which sends the data to random cut forest where the anomaly scores are created. This is the SQL code for the real time analytics:

```
1  -- ** Anomaly detection **
2  -- Compute an anomaly score for each record in the source stream using Random Cut Forest
3  -- Creates a temporary stream and defines a schema
4  CREATE OR REPLACE STREAM "TEMP_STREAM" (
5     "dateminute"   BIGINT,
6     "weekdayId"  INT,
7     "numberOfRequests"   INT,
8     "ANOMALY_SCORE"  DOUBLE,
9     "ANOMALY_EXPLANATION"  varchar(512));
10
11 -- Creates an output stream and defines a schema
12 CREATE OR REPLACE STREAM "DESTINATION_SQL_STREAM" (
13    "dateminute"   BIGINT,
14    "weekdayId"  INT,
15    "numberOfRequests"        INT,
16    "ANOMALY_SCORE"  DOUBLE,
17    "ANOMALY_EXPLANATION"  varchar(512));
18
19 -- Compute an anomaly score for each record in the source stream
20 -- using Random Cut Forest
21 -- this is where the anomaly detection happens. See https://docs.aws.amazon.com/kinesisanalytics/latest/sqlref/sqlrf-random-cut-forest-with-explanation.html
22 CREATE OR REPLACE PUMP "STREAM_PUMP" AS INSERT INTO "TEMP_STREAM"
23   SELECT STREAM "dateminute", "weekdayId", "numberOfRequests", "ANOMALY_SCORE", "ANOMALY_EXPLANATION"
24   FROM TABLE(RANDOM_CUT_FOREST_WITH_EXPLANATION(
25     CURSOR(SELECT STREAM "dateminute", "weekdayId", "numberOfRequests" FROM "SOURCE_SQL_STREAM_001"), 100, 256, 100000, 1, true
26   )
27 );
28
29 -- Sort records by descending anomaly score, insert into output stream
30 CREATE OR REPLACE PUMP "OUTPUT_PUMP" AS INSERT INTO "DESTINATION_SQL_STREAM"
31 SELECT STREAM * FROM "TEMP_STREAM"
32 --WHERE ANOMALY_SCORE > 3.0
33 ORDER BY FLOOR("TEMP_STREAM".ROWTIME TO SECOND), ANOMALY_SCORE DESC;
```

SQL code for real time analytics

Source: https://medium.com/@devfire/real-time-anomaly-detection-in-vpc-flow-logs-part-5-anomaly-detection-d1fc9b61baf8

As we can see from the comment in line 32 it would be possible to already define a threshold here. This would then mean, that only anomaly scores which are higher than that specified threshold are sent to the output stream. However, it is recommended to implement this threshold in the output lambda function, to keep the threshold logic separate from the creation of the anomaly scores. Furthermore it is a lot easier to version and backup the lambda function in case the threshold is changed over time than to maintain different versions of the SQL code in the Kinesis Data Analytics part.

Lambda: Kinesis output function

The output of the Kinesis stream is sent to the Lambda function `medium_bmw_kinesis_to_dynamodb_2`. This Lambda function just stores the results to another DynamoDB called `kinesis_bmw_anomaly_scores`. Obviously this last database table was just for the purpose of the project phase to collect all the anomaly scores and visualize them in a graph but it is not needed in a production setup (as indicated in the system architecture). Instead, in a production system this output function should contain the threshold for the anomaly score and then send a message or trigger a phone call if the threshold is

exceeded.

Results

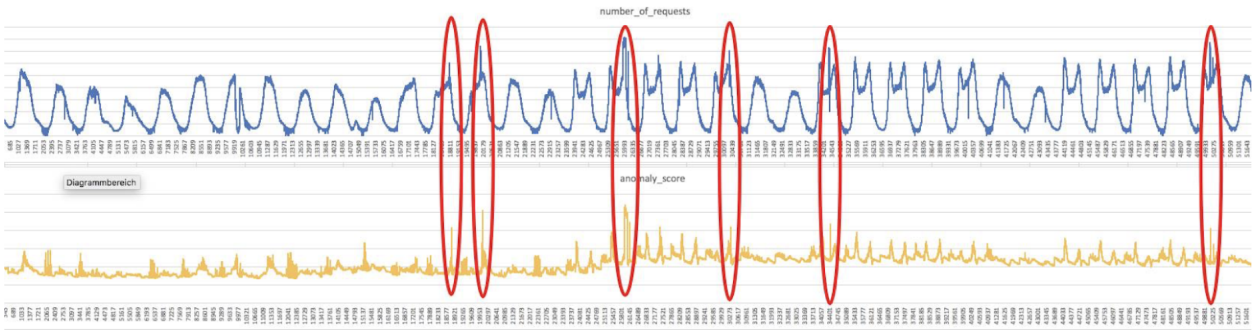


Figure 2.15: Visualisation of Kinesis Analytics Results

The blue graph at the top of figure 2.15 shows the requests to the the system per minute (from the one minute buckets). The orange graph at the bottom of figure 2.15 visualizes the anomaly score at that time. This means if we see an unexpected high peak or unexpected low drop in the first graph (number of requests per minute), in both cases we would expect a peak in the second graph (the anomaly scores). All in all this example does not show a perfect result (since we would expect our anomaly scores to be consistently low when there is no anomaly), however we have to take into account that this particular data set show the requests from December 21, 2018 until January 28, 2019, which means that it starts with the Christmas holidays. As we know, holidays to not coincide with the normal pattern of five working days followed by two weekend days which we see towards the end.

Furthermore, this is a series of 54,689 one minute buckets which is not a very long series to train on. Despite these factors of having limited and atypical data the high spikes in the anomaly scores (highlighted by the red ellipses) indicate that the approach in general is quite promising and that it can already detect anomalies.

Future work

The presented results are retrieved from feeding the data into the Random Cut Forest in the following format: date-minute data, the weekday ID (as an integer) and number of requests. To refine the system further it might be interesting to try different combinations of these three input variables to see which combination gives the best results. This way it could be determined if it decreases the performance if we leave out the information about the weekday completely for example.

Implementation guide (Kinesis Random Cut Forest)

This can be found in the appendix of this report (Appendix A) or online at https://gist.github.com/clemenspeters/8e9025e3bd71e9087df154fb06f96328

2.3 Prediction algorithms

As presented in the introduction (1.2.2), the team also compared different approaches of time series forecasting for the purpose of predictive auto-scaling of BMW servers. Whenever an amount of network traffic is predicted that lies above or far below of what the servers are able to handle, further resources shall automatically be allocated or released. In this way, the server costs can be minimized, while still ensuring a seamless experience for the customers. For this purpose, we again used the Mean Predictor (2.3.1), as well as a Deep Learning algorithm provided by AWS called DeepAR (2.3.2). Additionally, we introduce the Holt-Winter's Method for prediction (2.3.3).

2.3.1 Mean Predictor Model

Author: Marius Schidlack

Naturally, we designed the Mean Predictor from the previous section such that it can also be used for timeseries forecasting without any additional work required, by simply predicting the expected value for each point in a week.

2.3.2 DeepAR Model

Author: Alexandre Rozier

Presentation

We'll start the presentation of this algorithm by quoting AWS on DeepAR [6]:

> "The Amazon SageMaker DeepAR forecasting algorithm is a supervised learning algorithm for forecasting scalar (one-dimensional) time series using recurrent neural networks (RNN). Classical forecasting methods, such as autoregressive integrated moving average (ARIMA) or exponential smoothing (ETS), fit a single model to each individual time series. They then use that model to extrapolate the time series into the future."

Use-cases

The strength of this algorithm resides in the fact that it is able to learn simultaneously from different correlated time series (called `targets`) and their associated `dynamic features`, which are additionnal time series used as features during training. Please note that it can't be trained on streaming data.

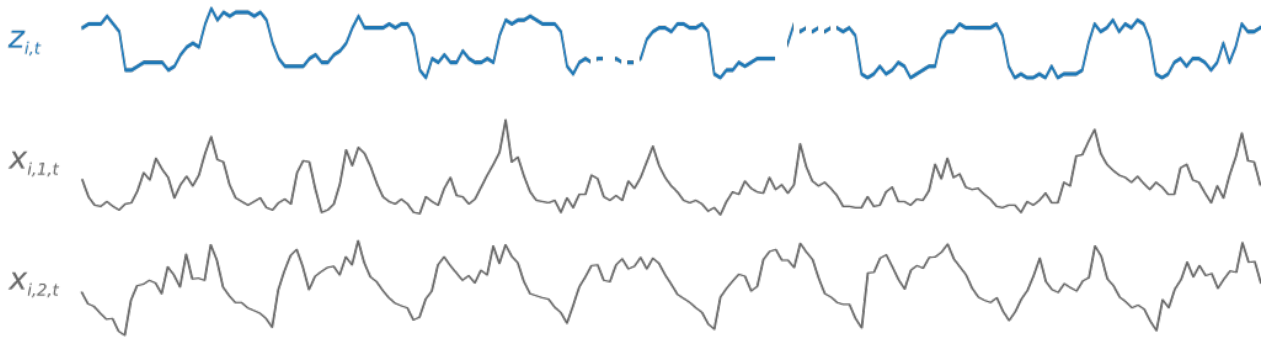


Figure 2.16: DeepAR learns to predict a target (in blue), possibly from dynamic features (in black)

To take a concrete example, let's say we wish to predict the requests hitting the BMW servers on a per-city basis. Our `targets` would be the weekly profiles of requests coming from each major city, and the associated `dynamic features` could for instance be the weekly profiles of oil prices and temperatures in those cities.

Expected Input

For the sake of clarity, below is an example of what data shape DeepAR expects. Each line corresponds to a `target` time series, and each `dynamic_feat` to a list of its associated dynamic features.

```
1 {"start": "2009-11-01 00:00:00", "target": [4.3, "NaN", 5.1, ...], "
    cat": [0, 1], "dynamic_feat": [[1.1, 1.2, 0.5,...]]}
2 {"start": "2012-01-30 00:00:00", "target": [1.0, -5.0, ...], "cat":
    [2, 3], "dynamic_feat": [[1.1, 2.05, ...]]}
3 {"start": "1999-01-30 00:00:00", "target": [2.0, 1.0], "cat": [1, 4],
    "dynamic_feat": [[1.3, 0.4]]}
```

As a further remark, one can point out that DeepAR supports unknown target values during training, which can be useful if we have an outage at some point cutting access to the data stream.

Performance on the BMW dataset

We tried three different approaches with the DeepAR algorithm, all having a common agreement: train on all the dataset but the last week, and test on the last week.
It is useful to remark that one could have removed the part of the data being the Christmas holidays, but we were curious about the ability of DeepAR to abstract these "outliers". We trained with the following parameters:

1. **Training without dynamic features**

   Idea:

   Training set on all data but the last week, performance evaluation on the last week, without any dynamic features.

   Input shape:

```
1          {"start": "2019-11-01 00:00:00", "target": [4.3, "NaN
               ", 5.1, ...]}
```

   Results:

   Remarkably, the algorithm performs quite well during the working days with a

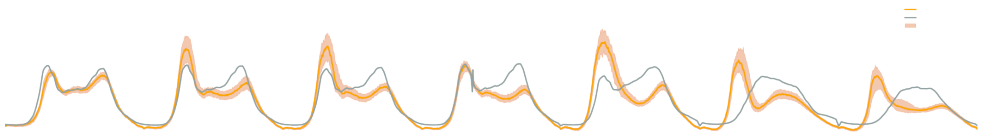


Figure 2.17: DeepAR Prediction trained without dynamic features.

   RMSE (Root Mean Square Error) of 95K. but is unable to learn the trends of weekends. This is quite interesting, since DeepAR is supposed under the hood to automatically encode additional temporal information such as the day of week. Maybe adding some hidden neurons would improve its ability to learn the weekend patterns.

2. **1-Week forecasting training with holidays features**

   Idea:

   Training set on all data but the last week, performance evaluation on the last week, with holidays dynamic features. When a day is a holiday, its associated dynamic feature is a one, else 0.

   We train the model to predict a full week , with a week of context.

   Input shape:

```
1          {"start": "2019-11-01 00:00:00", "target": [4.3, "NaN
               ", 5.1, ...], "dynamic_feat": [[0, 0, 1,...]]}
```

   Results:

   RMSE (Root Mean Square Deviation) of 72K

3. **1-Week forecasting training with holidays & weekends features**

   Idea:

   Training set on all data but the last week, performance evaluation on the last week, with holidays & weekends dynamic features. When a day is a holiday, its associated dynamic feature is a one, else 0. And similarly, when a day is a weekend, its associated feature is a one, else 0. We decided to add the weekends feature since in the first part we identified that DeepAR had trouble learning their pattern.

   We train the model to predict a full week , with a week of context.

   Input shape:

```
1          {"start": "2019-11-01 00:00:00", "target": [4.3, "NaN
               ", 5.1, ...], "dynamic_feat": [[0, 0, 1,...],[1, 1,
                0, 0, ...]}
```

   Results:

   RMSE (Root Mean Square Deviation) of 122K Adding the weekends dynamic features deeply worsened the model performance, and we think this is because it might conflict with what DeepAR does already perform under the hood.

This shows that adding extra dynamic features do not necessary improve this model.

4. **3H forecasting training with holidays & weekends features**

   Idea:

   Training set on all data but the last week, performance evaluation on the last week, with holidays & weekends dynamic features. When a day is a holiday, its associated dynamic feature is a one, else 0. And similarly, when a day is a weekend, its associated feature is a one, else 0.

   The main idea behind this is to have a higher hourly granularity during the forecast, this being useful for instance during predictive auto-scaling.

   Input shape:

```
1          {"start": "2019-11-01 00:00:00", "target": [4.3, "NaN
               ", 5.1, ...], "dynamic_feat": [[0, 0, 1,...],[1, 1,
                0, 0, ...]]}
```

   Results:

   RMSE (Root Mean Square Deviation) of 132K For some reason, this tuning of the algorithm did perform poorly. One of our guess is that the data aggregated in 5-min buckets is too broad, and that we should rather aggregate it in 1-min buckets. Also, it is possible that we are working with too little data as well.

Discussion and recommendations

As mentionned in AWS description, DeepAR outperforms traditional time series forecasting algorithms when the dataset comprises thousands of correlated time series. We tested it on a single time series coupled with only two dynamic features, but it already had promising results (see 2.17), improving continuously with the number of dynamic features.

However, even though DeepAR is supposed to generate additional time features such as day of week, it did not perform well on weekends, and we had to input these features dynamically. This leads us to believe that there is a bit of tweaking to do in order to obtain a foolproof model.

Due to the lack of geolocalized data, we tested this algorithm on a simple task, whereas it is designed to work on a lot more complex task. Thus, we highly recommend that you try it **when additional, geolocalized data will be available, such as traffic per world area. In this case, DeepAR could outperform traditional models and give BMW valuable insight**.

2.3.3 Holt-Winter's Method

Author: Marina Mursa

An intuitive assumption about the way one can predict the future is that the future should resemble the past experiences. So the most naive approach in forecasting is just to set the position of a future datapoint to the same place as the last observed one. In this section, we will talk in details about an approach that builds on this principal.

Theoretical background

One of common approaches used for forecasting, that is based on this historical principal, is called *exponential smoothing*. In a simple version of *exponential smoothing* we just copy and paste the last observed data. This however disregards historical data, except for the most recent ones. A more thoughtful approach would be to assign some weight to each data. The more recent a datapoint will be, the bigger its weight value.

Think about it for a second. Imagine a distribution in the form of a diagonal line, say

starting in the bottom left corner going to the upper right corner. Our human perspective can identify easily that the data has a *trend* of going up. If we just average all the datapoints and produce a mean prediction, the prognosis will be a line that goes straight in the middle of the graph. This is obviously not the result we would want.

This way, the simple exponential smoothing can not account for forecasting time series data that has a trend and/or a seasonal component. Luckily, this issue can be resolved by using an upgraded version of smoothing called Holt-Winters method [7].

It divides a time series Y_t in three parts: one related to its seasonality (s_t), one related to its trend behavior (b_t) and one for the residual part (l_t). For each of them, a simple EWMA is applied to predict a new value. A combination of these expressions is used to estimate Y_{t+1}. The following equations represent the HW computation:

$$\begin{aligned}
\hat{y}_{t+h|t} &= \ell_t + hb_t + s_{t+h-m(k+1)} \\
\ell_t &= \alpha(y_t - s_{t-m}) + (1-\alpha)(\ell_{t-1} + b_{t-1}) \\
b_t &= \beta^*(\ell_t - \ell_{t-1}) + (1-\beta^*)b_{t-1} \\
s_t &= \gamma(y_t - \ell_{t-1} - b_{t-1}) + (1-\gamma)s_{t-m}
\end{aligned}$$

Before explaining how Holt-Winter method was implemented, let's take a moment and think *why would we need to explore this solution.*

Use-case motivation

Going back to the use case of this project, the implementation of this algorithm would, in theory, be able to generate a prediction, as well as extract a series of interesting knowledge about historical data.

First of all it would be able to find an existing *trend* in the data. This trend will probably depend on the rate of new cars joining the ConnectedDrive network over time, or on the

growing popularity of certain services (that would increase the number of daily request from a car) and it will be best visible after a large amount of data is collected.

Secondly, it should pick up on any *seasonal components*, if they exist, and give a hint towards existing outside factors that can influence the traffic. A hypothesis can be the fact that in summer, when kids do not have school and lots of people chose to go on vacations, the number of cars on the streets is smaller compared to winter. Once again, having enough data (for at least a year), these type of hypothesis can be tested by training the model and decomposing the data.

Data testing

To see if such an algorithm is applicable to a dataset, first of all we can run a decomposition function that would try to split a batch of data into the 3 equations mentioned above.

```
1 decompfreq = ((24*60)//5)
2 sm.tsa.seasonal_decompose(train, freq=decompfreq).plot()
```

Listing 2.3: Data decomposition

In figure 2.18 (Part.1) you can see the results from decomposing the BMW test dataset. If we analyse the graphs, we can observe that the model is doing a fairly good job at handling the data. We purposely included the anomalous first two days (21-22 December) into our training data to observe how well does the model pick up on the existing trend as well as how easy it is to through it off. We can observe that the trends are well preserved, although their specifics are dimmed down.

In comparison, the seasonal graph does not do so well, while it does detect a daily pattern, it only plots almost Gaussian distribution of daily data. This is due to the poor quality of data, that includes a holiday period, where the rush hour pattern is non-existent. Another factor that diminishes the pattern is the fact that the model averages over weekends, where again the double-peaked pattern disappears. If we exclude this holiday period, we can see a

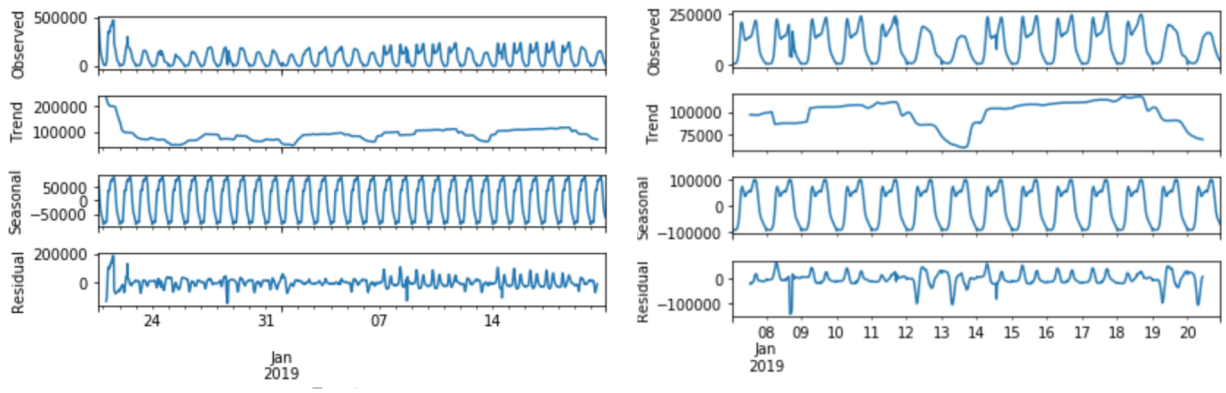


Figure 2.18: Data decomposition into Trend, Seasonality and Residual using Holt-Winter Method

significant improvement in the pattern (see fig. 2.18 Part 2).

Now that we know that our data is suitable for a Holt-Winter's Method analysis (because it performed the decomposition successfully), let's see how the model can be implemented.

Model Implementation

First of all we should go back to our data. The imputed format does not matter much as long as 2 dimensions of the data can be extracted, `count` and `datetime` (or other similar inputs).

Next step is to divided the data into training and test data. As mentioned in the previous sections, the agreement for the prediction algorithms was to leave the last week for testing, otherwise, around 0.2 parts of the data can be left out for model validation.

After having the two sets, no further reprocessing is required.

The *statsmodel* module in Python has a pre-built function to do the so-called **Triple Exponential Smoothing**. In this method, we can implement both the *additive* and *multiplicative* technique. The additive method is preferred when the seasonal variations are roughly con-

stant through the series, while the multiplicative method is preferred when the seasonal variations are changing proportional to the level of the series.

```
model = ExponentialSmoothing(train, seasonal_periods=seasonl,
    seasonal='mul').fit()
pred = model.predict(start=test.index[0], end=test.index[-1])
```

Listing 2.4: Exponential Smoothing method

The constants α, β, γ (that were mentioned in the formulas presented in the sections above) - can be estimated (usually through a trial and error process known as *fitting*) and can be left for the algorithm to estimate or manually set in the .fit()[7] method as following:

```
ExponentialSmoothing.fit(
    smoothing_level=None, smoothing_slope=None,
    smoothing_seasonal=None, damping_slope=None,
    optimized=True, use_boxcox=False,
    remove_bias=False, use_basinhopping=False,
    start_params=None, initial_level=None,
    initial_slope=None, use_brute=True
)
```

Results

Bellow you can see the results of running such an algorithm. Although the **RSME** is about **48K**, which is a relatively good performance, and as seen in fig. 2.19 the prediction results are close to the reality, they still have the problem of recreating the double peaks pattern. This can be explained by the fact that the algorithm has no means of separating the weekdays from weekends (as DeepAR does), and end up averaging all days together.
In addition, because the model puts together three distributions (trend, seasonality and residual) every time it trains, such an algorithm requires a lot of computational power.

This being said, we need to remember that our motives for trying out this algorithm was to discover new knowledge about data, besides being able to generate predictions. As we saw

[7] https://www.statsmodels.org/dev/generated/statsmodels.tsa.holtwinters.ExponentialSmoothing.fit.html#statsmodels.tsa.holtwinters.ExponentialSmoothing.fit

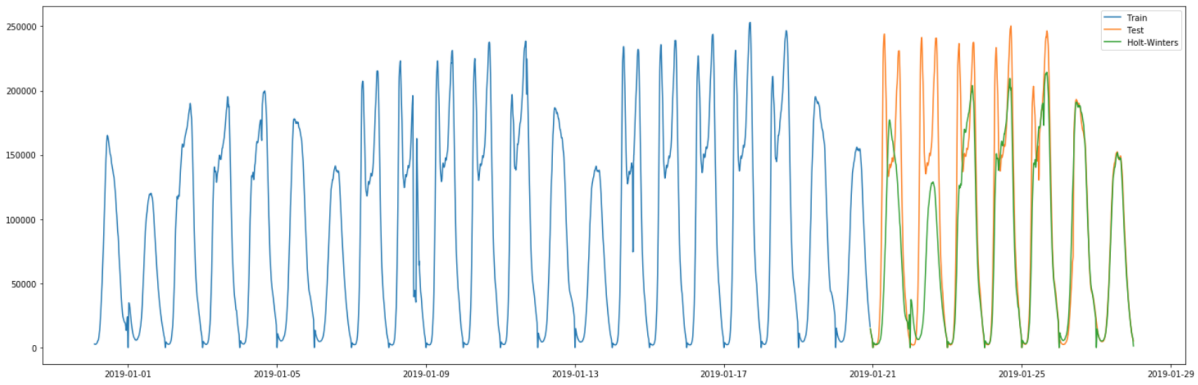


Figure 2.19: Results of Holt-Winter's prediction model

from the decomposition it only detects the already known day and week pattern. This is of course due to a very small input of data (3 weeks).
If we would gather a more considerable amount of training information, let's say for a year, this model would certainly have more secrets to discover.

2.4 Data Management

Author: Ali Karaki, Jacek Janczura
The following chapter describes different approaches of connecting the above mentioned models with the data sources. We also discuss the process of creating a work-flow between Data-Model-Results in Notifications.

2.4.1 Mean Predictor V1

In this chapter the first prototype of the pipeline. that uses the mean predictor algorithm, will be described. In the pipeline to detect anomalies: lambdas, models, and S3 buckets, to create the work-flow of sending data to a model and retrieving back results, were combined. Working with lambdas, layers, dependencies and overall usage of a single lambda function

across several users was a tricky process.

A great challenge was discovering usage of layers - tool that enables easy possibility of importing several additional python libraries to lambdas. Thanks to the layers, lambdas development process can be accelerated and shared among other developers. AWS built in libraries no longer limit us and lambdas code can be viewed in the AWS console. Layers, even though they are very powerful, are badly documented and it took us a great deal of time to discover the correct usage.

Mean Predictor was the simple model we first created, and the lambdas used to make it fully functional revolved around accessing an S3 bucket with the correct permissions and policies, retrieving data or modifying data in said bucket, then sending it to the model. Input data format is JSON and the results were either JSON or in CSV format, according to our needs we changed the formats on demand.

Two main functions were used here, `Model_Data_Join.p` and `AnomalyDetection.py`. We also used a function `alertDemo.js` for Slack integration and alerts. Our first phase of trying to connect all of the above mentioned elements together was challenging. Each function used different policies and dependencies, and AWS has some limitations with uploading .zip files larger than a specific size. We also created Layers, acting like a placeholder for all dependencies that are being used. In addition to those, we also made an API Gate-Away named `RCF_Data_Join_SageMaker` that gives us the possibility to call the model and send some data to it, without actually being in AWS console at all. We used PostMan for the requests.

Model_Data_Join.py can be found in the `/lambdas` folder in the github repository. This function's main job is to send data to the model. Using a specific prefix directory, we let the function access our data bucket and look up all keys inside it. Sent data is going to be tested for anomalies, so we couldn't just send a VPC-FlowLog to the model. Instead, we created a file reader, where we read all the keys inside the directory (Keys are FlowLogs), we parse a

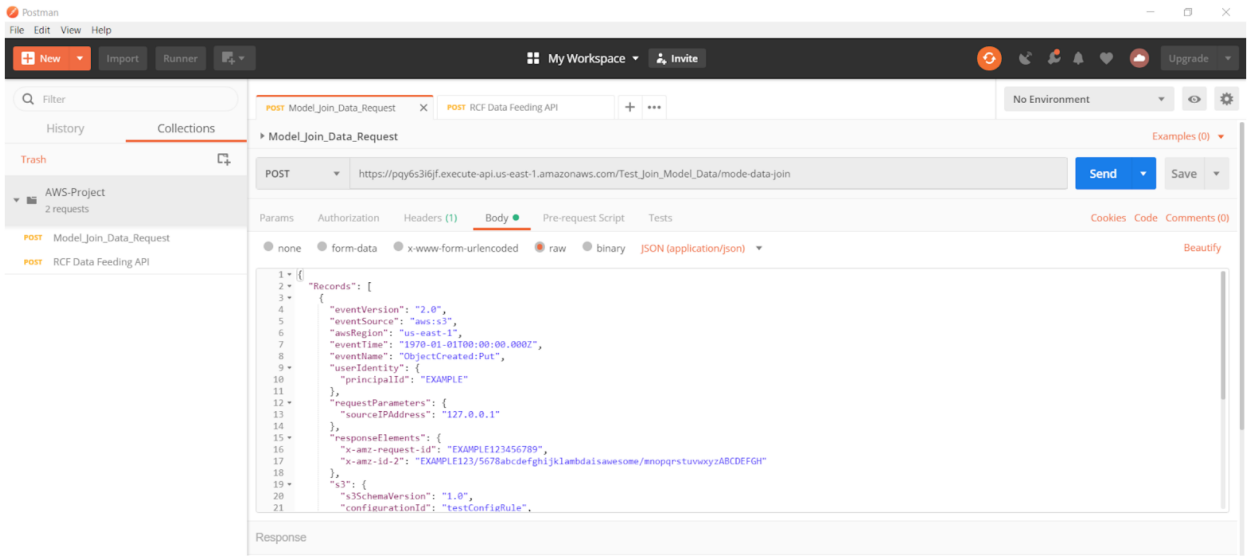


Figure 2.20: Example of API Gateway using Postman

small JSON file with index and the number of lines inside them. The number of lines equals the number of request inside each single Flowlog. This was the most convenient way to send requests number to the model using FlowLogs (Fig. 2.21).

After the data has been parsed from the keys, we get a list, converted to a JSON file. This file is later dumped inside the bucket in order to be kept as a history of data that has been used. This process also involves sending the data to the model, after parsing it to JSON format. We used `invoke method` provided from AWS to achieve this. The method takes another lambda function `AnomalyDetection` and the data file as a variable `(json_data)`. Below you can see the code for this.

After sending the data to `anomalyDetection` lambda, we parse the event back to CSV format for the model to understand it. We separate each value by a new line. Results from the model are then parsed and passed through a prediction method. This method detects which values where identifies to be TRUE as anomalies. This result is sent to our AWS channel using an SNS topic `(arn:aws:sns:us-east-1:746022503515:api-test)` created to queue all notifications. Since this was our first demo, we didn't enable automatic triggering with

```python
def lambda_handler(event, context):

    #Grab the directory with all needed keys (Key is a file)
    list_keys('fog-bigdata-logs', '/streamed2018/12/14/')
    i = 1
    j = 1
    print('Number of Files in this directory = ', len(keys_list))

    #Loop to read each key
    for x in keys_list:
        #Connect to S3 and get File name then read it
        file_obj = event["Records"][0]
        print("\n########\n\n",i,"- Filename: ", x)
        #Grab a single key and read it's content
        fileObj = s3.get_object(Bucket = "fog-bigdata-logs", Key='/streamed2018/12/14/' + x)
        file_content = fileObj["Body"].read().decode('utf-8')
        #Number of requests is the number of lines in this file
        requests = len(file_content.split('\n'))
        print("Number of requests = ",requests)
        if i<32:
            json_list.append(tuple((i, requests)))
        i += 1
        # Uncomment this to get file content
        # print(file_content)

    #Json dump
    json_result = dump_to_json(json_list)
    # return dump_to_json(json_list)
    return json.loads(json_result)

#Function to list keys in a specific bucket according to a prefix
def list_keys(bucket_name, prefix):
    for key in s3.list_objects(Bucket=bucket_name, Prefix=prefix)['Contents']:
        if(key['Key']).endswith('.bucket'):
            keys_list.append(key['Key'].split("/")[-1])
```

Figure 2.21: Lambda Key Reader

every new data file, instead, we created test events with the needed buckets and triggered them manually. To make it work properly, you just need Flow-Logs inside the directory (as briefly shown in Fig. 2.22).

```python
#Function responsible for sending a list of data, converted to json, as a request to the other function that calls the model
def dump_to_json(list):

    #change region_name as per the destination lambda function region
    invokeLam = boto3.client("lambda", region_name="us-east-1")

    data = {}
    data["data"] = list
    json_data = json.dumps(data)

    #Write to json file

    file_name = 'json_dumps_{}.json'.format(timestr)
    # file_name = 'json_dumps.json'
    lambda_path = "/tmp/" + file_name
    s3_path = "json_dumps/" + file_name
    s3 = boto3.resource("s3")
    s3.Bucket('fog-bigdata-logs').put_object(Key=s3_path, Body=json_data)

    #This will invoke @Marius function, Anomaly Detection.
    resp = invokeLam.invoke(FunctionName = "anomalyDetection", InvocationType = "RequestResponse", Payload = json_data)
    # print(resp["Payload"].read().decode())

    print (json_data)
    return(resp["Payload"].read().decode())
```

Figure 2.22: JSON Dump code

2.4.2 RCF (Fixed Data)

RCF Random Cut Forest is an unsupervised algorithm for detecting anomalous data points within a dataset and was used in a SageMaker as our second trial model. It was described in the sections above.

The data connection with this model was more challenging. Even though the model took normal JSON format, we had to separate the values and parse the results several times to make sure they are ready for a clean execution. The model takes a list of data points, then returns a result which is `Scores_Array`. These scores have to be calculated again using a specific function to determine if they are anomalies (above or below the RCF threshold). Based on those values, we trigger an alert that sends notifications to a channels. This procedure was changed to be almost fully automatic. We added a trigger to the functions `rcf-sagemaker-testdata` with a suffix (.csv) and `objectCreated` event over an S3 bucket, which triggers the whole work-flow by itself. An API gateway `RCF_API` was also created for

```
def lambda_handler(event, context):
    print("Received event: " + json.dumps(event, indent=2))

    #Receive the event (data from first lambda function) and convert to json readable format
    data = json.loads(json.dumps(event))
    payload = data['data']
    # Convert the data to csv format
    payload = list(map(lambda d: str.join(',',map(str,d)), payload))
    payload = str.join('\n',payload)

    #call the model
    response = runtime.invoke_endpoint(EndpointName=ENDPOINT_NAME,
                                       ContentType='text/csv',
                                       Body=payload)

    # Parse the csv response from the model, which is the result
    res_str = response['Body'].read().decode("utf-8")

    #Run predictions calculations
    predictions = list(map(lambda st: st == 'True',str.split(res_str,'\n')[:-1]))

    #Assign it to SNS topic for notifications publishing
    sns = boto3.resource('sns')
    topic = sns.Topic('arn:aws:sns:us-east-1:746022503515:api-test')

    response = topic.publish(
        # TargetArn='arn:aws:sns:us-east-1:746022503515:api-test',
        Message=json.dumps(dict(
            default=str(predictions)
        )),
        Subject='alert',
        MessageStructure='json',
    )

    return predictions
```

Figure 2.23: Lambda precision

the same purpose of our first model.

Two S3 buckets were involved in this process, `rcf-sagemaker-finished` and `rcf-sagemaker-testdata`. A small overview of the work-flow is described below:

1. Make sure the model endpoint is working properly (preliminary checks per usual)
2. Upload data to 1 bucket `rcf-sagemaker-testdata`
3. This upload to S3 bucket automatically triggers using SNS topic all subscribed lambda

functions.

4. 2 functions are then triggered. The data from S3 bucket is read, prepared and send to the model. Since the model is already feed with a new data, the input data from the initial S3 bucket are moved to the finished sets to `rcf-sagemaker-finished` S3 bucket.

5. Results and especially the anomaly detection, trigger the notification functions which inform the user about the anomalies.

Used functions The whole process involved 2 functions, `RCF_Data_Join_SageMaker.py` and `RCF_Anomaly_Model_2.py`. The first function is triggered by file upload to the S3 bucket and reads uploaded data. After parsing the data the function sends the filtered data to the second function, where we call the model for analysis. All policies, notes, used variables, endpoint names etc. can be found in github repository under `Notes.txt`. The function reads all files in the S3 bucket and lists them for logging purposes. Once we upload a file, it separates it into 2 versions. The data is usually DATE SCORE format. So we create 2 lists, 1 with both values and 1 that has only the score data. We do this because we want to preserve the date as time-stamp history for the model. Once we have the parsed data, the process of sending this data starts. The smart thing about this is that we used AWS functions to clear our buckets. The data gets sent from `rcf-sagemaker-testdata` bucket to `rcf-sagemaker-finished` bucket, with the name changed to include time of the data testing and date as shown in the Fig. 2.24.

We then proceed to call `dump_to_csv`, where we send the data to the model including the date time-stamp, and also create a `csv_dump` folder inside the finished bucket for testing and debugging purposes.

The model function, `RCF_Anomly_Model_2.py` is then invoked with the needed json data. And as a result of some time consuming calculations in this function, we had to make the run-time of lambda up to 3 minutes to avoid any unwanted failures.

```
    #Split the whole file into lines, in a list
    sum_values_list = file_content.splitlines()

    #Remove the date from each element inside the list
    sum_list(sum_values_list)
    del_date(sum_values_list)

    # i += 1
    #Copy the file to another bucket after we finished working on it
    copy_to_bucket('rcf-sagemaker-testdata','rcf-sagemaker-finished', keys_list[0])

    #Json dump
    json_result =  dump_to_csv(del_date_list)

    # return dump_to_json(json_list)
    return json.loads(json_result)


#Remove date from elements taking comma as a delimeter
def sum_list(list):
    for item in list:
        # csv_list.append(item.split(",")[-1])
        csv_list.append(item)

#Remove date from elements taking comma as a delimeter
def del_date(list):
    for item in list:
        del_date_list.append(item.split(",")[-1])

#A standarized way to copy files from 1 bucket to another.
def copy_to_bucket(bucket_from_name, bucket_to_name, file_name):
    copy_source = {
        'Bucket': bucket_from_name,
        'Key': file_name
    }
    s3_r.Object(bucket_to_name,"Data Finished at {} --> ".format(timestr) + file_name).copy_from(CopySource=bucket_from_name + "/"
    # s3_r.Object(bucket_to_name, file_name).copy(copy_source)
    s3_r.Object(bucket_from_name,file_name).delete()
```

Figure 2.24: The moving of data from testing buckets to finished testing data

After the json event is received, we parse it back to the correct csv format and send it to the model using `invoke_endpoint` method. The result is a huge CSV response with all the values we need to calculate with RCF for anomalies.

The final cut-off score is calculated with the following equations: Based on the calculated RCF Threshold we compare the values based on the input data and mark those over the threshold level as anomalies. As soon as anomaly is detected notification process starts.

Once we have everything calculated and compared, the results are sent to another part of the function where we start sending alerts and notifications. For this we created a specific Slack App & another function that handles all the notifications. This can be found in a detailed section down below under Notifications Functions.

```python
#Generates json files and sends them to a lambda function containing another model call
def dump_to_csv(list):

    #change region_name as per the destination lambda function region
    invokeLam = boto3.client("lambda", region_name="us-east-1")

    data = "\n".join(list)

    #Write to the list to a csv file and store it inside a bucket
    file_name = 'csv_dumps_{}.csv'.format(timestr)
    lambda_path = "/tmp/" + file_name
    s3_path = "csv_dumps/" + file_name
    s3 = boto3.resource("s3")
    s3.Bucket('rcf-sagemaker-finished').put_object(Key=s3_path, Body=data)

    #This will invoke @Nursulta's function, RCF Anomaly Detection.
    resp = invokeLam.invoke(FunctionName = "RCF_Anomaly_Model_2", InvocationType = "RequestResponse", Payload = parse_to_json(csv_l
    # print(resp["Payload"].read().decode())

    # print ("\n\nAll Sum Data from CSV function:\n", data)
    return(resp["Payload"].read().decode())

#Request to model needs to be a json, so this function with parse the data to json before sending it
def parse_to_json(objects):

    # objects = [float(i) for i in objects]
    data = {}
    data["data"] = objects
    json_data = json.dumps(data)
    # print (json_data)
    return json_data
```

Figure 2.25: Invoking RCF-Anomly-Model-2.py

```python
#Calculate cut-off final score
scores_array = numpy.array(score_array)
score_mean = scores_array.mean()
score_std = scores_array.std()
score_cutoff = score_mean + 3*score_std

cut_off_final_score=('Cut-Off final Score:',score_cutoff)
```

Figure 2.26: Cut-off score calculator

The graph generation was a bit of a complicated manner, since we can't send direct images from lambda to slack without doing a workaround. We had to store the results in a memory buffer, and read that buffer to send it. Notifications being sent had the anomalies, a graph with the data + threshold + anomalies detected in a detailed form.

Using panda, we saved data points in an X (**Date Stamp**) and Y (**Scores**) lists after reading

```python
#Comparing the final cut-off scores with a list of data, these are the final cut-off scores
def compare_cut_off(response_here, list_here):
    print("Final Cut-Off Scores with TIMESTAMP + Score: \n")
    i = 0
    string_2 = ""
    for items in list_here['data']:
        if float(score_array[i]) > response_here:
            # print(str(items.split(",")[0]), float(score_array[i]))
            string_2 = "---".join([str(items.split(",")[0]), str(score_array[i])])
        i += 1
    return ("Anomalies found --> TIMESTAMP + Score:",string_2)

#Used for debugging purposes as well. It prints the received payload to check for errors in logging
def print_payload(payload):
    # print(payload)
    for item in payload['data']:
        print(str(item.split(",")[0]), float(item.split(",")[1]))
```

Figure 2.27: Cut-off comparison

the file from lambda's tmp directory and generate the plot. Then sent to slack with a special

```python
#Plotting the graph and sending it to Slack Channel
def plot_send(dataFile, anomaly_score, array_results):

    x=[]
    y=[]
    i=0
    data = json.dumps(dataFile['data'])

    #Read the json as a csv
    df=pd.read_json(data)
    df.to_csv('/tmp/results.csv', index=False, header = 0)

    #Grab the csv from tmp in lambda then parse the data into x and y array points
    with open('/tmp/results.csv', 'r') as csvFile:
        reader = csv.reader(csvFile, delimiter=',')
        for row in reader:
            # print((row[0].split(',')[0]))
            #Apped time to X
            x.append(datetime.strptime((row[0].split(',')[0]), '%Y-%m-%d %H:%M:%S'))
            #Append data point to Y
            y.append(float((row[0].split(',')[1])))
            #if point is anomaly, then plot it
            if array_results[i] > anomaly_score:
                plt.plot([datetime.strptime((row[0].split(',')[0]), '%Y-%m-%d %H:%M:%S')], [array_results[i]], marker
            i += 1

    plt.plot(x,y)
    #Horizantal line for Anomaly
    plt.axhline(y=anomaly_score, color='r', linestyle='-', label='Threshold')

    plt.title('Data from CSV: TimepStamps and Scores')

    plt.xlabel('TimeStamps')
    plt.ylabel('Scores')
```

Figure 2.28: Plotting the anomaly graph

`payload_2 param`, that holds our channel name, title and some other needed variables for better readability. In general, this all happens in an automatic form.

2.5 Notification functions

Author: Jacek Janczura

Notification part is a crucial part of our project as after collecting the data, preparing it and analysing it by using machine learning algorithms, we needed to find a way to communicate the results to the user. In this paragraph we will describe the motivation and ways of informing the user about the anomalies and predictions.

2.5.1 Motivation

To notify the user in order to react to main events such as anomaly detection, we created notification functions. Firstly we call the user by triggering ChatBot API. Then we send a notification to the Slack channel with a graph and a highlighted anomaly to allow the user to react faster.

User story

We decided to create a user story, where the user, who would be notified in case of an alert, is one of the administrators - Bob. Bob and his team use slack in their work, just like we do in this project.

In case of the anomaly detection we trigger ChatBot API to call Bob and notify him about the anomaly. To let Bob take his decision faster and - in case of an emergency - escalate the problem (or - in case of false detection - skip it), we decided to show him the graph, where he can visually assess the problem. Since Bob and his team use Slack, we decided to send them a notification to a special Slack channel as well.

To accelerate the decision-making process, even before logging in to the consoles, cloud watches etc. we decided to send Bob a part of the graph with the prediction and the marked anomaly. In such a way, Bob or his colleagues receive a straightforward information of what

is currently happening. The example of notification massages sent to the slack channel is depicted in a fig. 2.29.

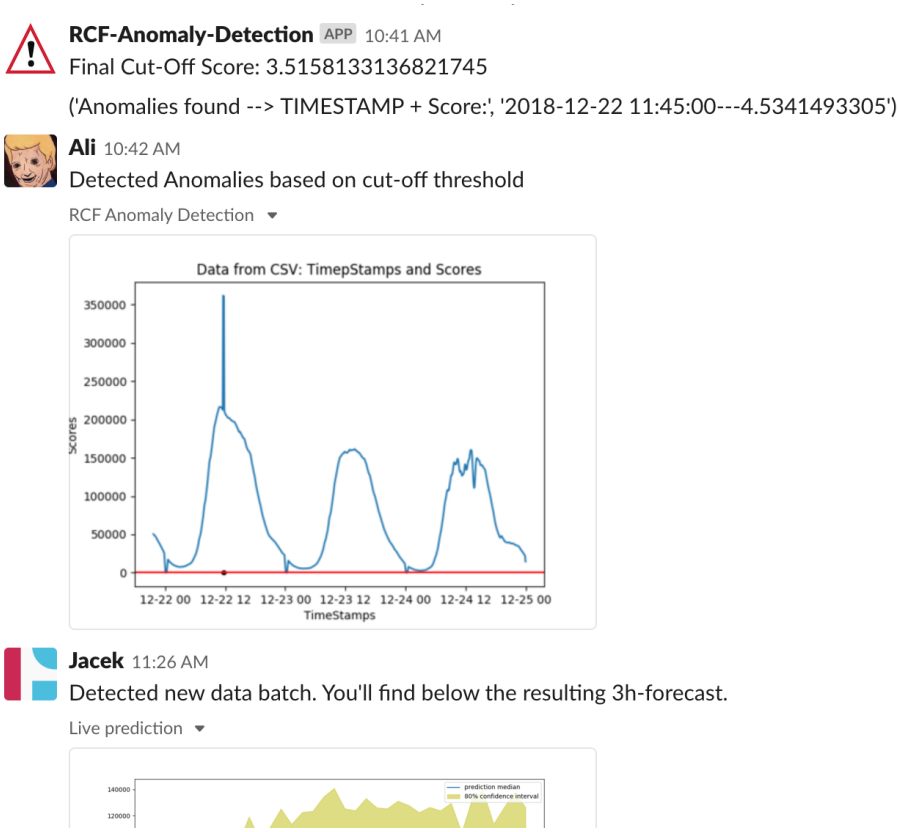


Figure 2.29: Example of anomaly notification

2.5.2 Methods to notify the user about detected anomalies

- **ChatBot API notification** - To notify the ChatBot API in case of an anomaly we send REST POST message to the ChatBot API URL, as shown in an example in fig. 2.30. That POST triggers ChatBot, which calls the user.

```
function notifyChatBot(url, message) {
  const headers = {
    'Content-Type': 'application/json',
    'Authorization': 'Bearer xyz'
  }
  const request = {
              status: 'firing',
              description: message,
              priority: 'high'
          }

  console.log("This post will call our friends at ChatBoT: %j", request);
  axios.post(url, request, {headers: headers})
  .then(function (response) {
    console.log('Alles Gut!');
    console.log(response);
  })
  .catch(function (error) {
    console.log('Oh no something went wrong!');
    console.log(error);
  });
```

Figure 2.30: Example of ChatBot API notification

- **Slack channel metrics** - To inform the user about the exact metrics and data we send the message to the slack channel using Amazon Simple Notification Service topic.

```
1       #Send Slack MSG
2       sns = boto3.resource('sns')
3       topic = sns.Topic('arn:aws:sns:us-east-1:746022503515:RCF-
            SageMaker')
4
5       response = topic.publish(
6           Message=str("Final␣Cut-Off␣Score:␣{}".format(score_cutoff)
                ),
7           Subject='#aws',
8           MessageStructure='text/plain',
9       )
```

Listing 2.5: Sending message to slack channel using SNS topic

- **Slack channel graph** - Unfortunately sending the graph to the slack channel method using SNS topic did not work correctly. In order to show the graph using AWS lambda function we needed to perform the following steps:

  1. Plot a new graph in memory.
  2. Save that graph to a buffer and go to the beginning of a buffer

     ```
     buf = io.BytesIO()
     plt.savefig(buf, format='png')
     buf.seek(0)
     ```

  3. Add a buffer to a structure

     ```
     my_file = {
     'file' : ('./buf.jpg', buf, 'png')}
     ```

  4. Add that structure to the REST POST and post that file to the slack channel.

     ```
     r = requests.post("https://slack.com/api/files.upload",
         params=payload, files=my_file)
     ```

These simple notifications are an effective way to keep the user up to date with the framework.

2.6 Project delivery

Author: Alexandre Rozier

After six months of intensive work, we ended up with a pretty diverse code base that we needed to unify. We had various models, lambda functions, EC2 instances, roles and policies composing our processing pipeline, and hand them in the most convenient way.

2.6.1 Terraform

The source code for this part is located in `terraform/`.

We chose to use the infrastructure-as-code deployment tool Terraform, with a few quirks that are presented below.

Terraform uses a `state` to keep track of the state of managed cloud components, and perform its changes. This file is created locally on the machine running the terraform commands, but problems arise when two or more people try to concurrently run terraform scripts at the same time : inconstistencies could arise from such situations, which is to avoid at all costs.
To deal with this problem, we added the following piece of code:

```
1
2  terraform {
3   backend "s3" {
4     bucket         = "fog-bigdata-terraform-backend"
5     dynamodb_table = "terraform-state-lock-dynamo"
6     key            = "backend/terraform.tfstate"
7     region         = "${var.region}"
8     encrypt = true
9   }
10 }
```

Which hosts this `state` in a remote s3 bucket, allowing thanks to a locking mechanisme several people to work concurrently on the terraform files.

2.6.2 Lambda functions deployment

AWS has recently released a lambda function management tool called SAM, but unfortunately Terraform does not support it yet, so we had to package our lambda functions in ZIP files (ref) and use Terraform's `aws_lambda_function` to deploy them.

We had to overcome some hurdles, which are built-in AWS limitations regarding lambda functions:

- Zipped deployment package size limited to 50MB
- Unzipped deployment package, including layers, limited to 250MB

Since we are using the Matplotlib and Pandas librairies, we ended up with deployment packages weighting more than the AWS limit.
That's why we came up with the idea of creating our own layers (thanks to `aws_lambda_layer_version`

), and linking our lambda functions to those layers. Thus, we end up with ligthweight lambda functions (1MB), and also reusable layers.

2.6.3 IAM roles and policies

Every AWS service needs the correct policies to access the other components it works with. For instance, our lambda functions need to access AWS S3 and AWS SageMaker, and this is fortunately well supported by terraform.
The code below creates:

- A role that can only be assumed by AWS Lambda services
- Two policies giving full access to S3 and Sagemaker, attached to this previous role.

```
resource "aws_iam_role" "lambda_sm_s3_role" {
  description="Role giving lambda full access to S3 and SageMaker"
  assume_role_policy = <<EOF
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "",
      "Effect": "Allow",
      "Principal": {
        "Service": "lambda.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
EOF
}

data "aws_iam_policy" "AmazonS3FullAccess" {
  arn = "arn:aws:iam::aws:policy/AmazonS3FullAccess"
}

data "aws_iam_policy" "AmazonSageMakerFullAccess" {
  arn = "arn:aws:iam::aws:policy/AmazonSageMakerFullAccess"
}

resource "aws_iam_role_policy_attachment" "s3-full-access" {
  role       = "${aws_iam_role.lambda_sm_s3_role.name}"
  policy_arn = "${data.aws_iam_policy.AmazonS3FullAccess.arn}"
}
```

```
32
33 resource "aws_iam_role_policy_attachment" "sm-full-access" {
34   role       = "${aws_iam_role.lambda_sm_s3_role.name}"
35   policy_arn = "${data.aws_iam_policy.AmazonSageMakerFullAccess.arn}"
36 }
```

2.6.4 EC2 Instances

As previously mentioned in 2.1.4, we need to spin up an EC2 instance, but also upload and run a script generating a fake stream of data to feed our processing pipeline.
Since Terraform is only an infrastructure management tool, we needed an additional provisioning tool for this task. We went with Ansible, since it's pretty straightforward to use.
Thus, terraform calls a specific command after the creation of the EC2 instance, which runs an Ansible playbook (in `utils/ec2-provisioning.yml` charged with running the script.

Ansible provisioning

Since Terraform allows us to run specific scripts after provisioning of a resource, we used this hook to trigger an Ansible job dealing with the provisioning of our EC2 instance, and a few points are worth mentioning:

- Since we could want to SSH to the EC2 instance, we also used Terraform to manage an `aws_key_pair`, which is the public part of an SSH key pair of our choice. The private part is hosted on the machine running the Terraform script, and of course not commited to the source code.
- When considering the followinng piece of code :

```
1          provisioner "local-exec" {
2              command = "rm ansible-hosts.ini &&
3              echo \"${self.public_ip} ansible_user=ec2-user\" >>
                   ansible-hosts.ini
4              && ansible-playbook ec2-provisioning.yml --private-key
                    ~/Downloads/default-vpc-access.pem -i ansible-
                   hosts.ini"
5          }
```

We see that we have to first append to `ansible-hosts.ini` the public IP of our EC2 instance. This is because Ansible needs to know the IPs of its managed servers, in order to provision them.
Also, the `--private-key` argument should be changed to point to the location of the private key used to SSH to the server.

2.6.5 S3 Notifications & SNS Topics

Our lambda functions are triggered by SNS Topic events, monitoring uploads on specific endpoints of S3. We need to create those topics, and also the S3 notifications that send events to those, in order to have a working pipeline.
This is handled by Terraform, with the `aws_s3_bucket_notification` and `aws_sns_topic` directives, that take care of spinning up those services with the correct roles and policies.

2.6.6 S3 Buckets

The management of S3 bucket with Terraform can be tricky, since calling `terraform destroy` will delete all existing buckets and get rid of all our data! That's why we didn't import the bucket holding all our data to Terraform.

Instead, we create a new bucket called `sanitized-datasets`, which will after creation be populated by a script (`data-preprocessing.py`) with subfolders, each containing our data correctly formatted for each algorithm. For instance, `s3://sanitized-bucket/deep_ar/` will contain data with a DeepAR-compatible shape. This is implemented by using the `provisioner` parameter, which allows us to run local provisioning scripts after the creation of a resource. In this case, the provisioner takes care of populating the `sanitized-datasets` bucket with correctly-shaped data.

2.6.7 What was not covered

Even though pretty much everything is managed by Terraform, because of the high velocity of AWS development, some features were missing.

- Our lambda functions are usually triggered by SNS Topic events, but unfortunately this is not managed yet in Terraform. As a consequence, one has to add those triggers manually (on the AWS Lambda web editor).
- Also, SageMaker support is almost inexistent. That's why we had to leverage Terraform's `Provisioners` in order to train & deploy our models. If you consider the following code:

```
1       // Train a MeanPredictor model and export it as endpoint
2       provisioner "local-exec" {
3         command = "python ../models/mean_predictor/train_deploy.py
              --trainpath s3://${aws_s3_bucket.datasets.bucket}/rcf/
             data/train/data.csv --role ${aws_iam_role.sm_role.arn}
             --freq ${var.data_aggregation_frequency}"
4       }
5
6       // Train a DeepAR model and export it as endpoint
7       // WARING: takes ~ 2 hours
8       provisioner "local-exec" {
9         command = "python ../models/deep_ar/train_deploy.py"
10
11        environment {
12          BMW_DATA_BUCKET       = "fog-bigdata-bmw-data"
13          SANITIZED_DATA_BUCKET = "${self.bucket}"
14          SAGEMAKER_ROLE_ARN    = "${aws_iam_role.sm_role.arn}"
15          ENDPOINT_NAME         = "${var.deepar_endpoint_name}"
16          DATA_FREQUENCY        = "${var.
                data_aggregation_frequency}"
17        }
18      }
```

  The `local-exec` parts are the lines taking care of training and deploying our models. They are executed after provisioning of the S3 bucket holding all the correctly-shaped data.

Chapter 3

Project management techniques

Author: Marina Mursa

This chapter will give an overview to the ways this project was managed internally, as well as by the supervisors. It includes a detailed introduction to the Kanban - management tool, followed by complementary techniques used by the team to further improve the working process.

3.1 Kanban

For internal management of the team, based on the suggestions of our industry partners, it was decided to use an Agile product management technique - Kanban. It emphasizes on a continuous delivery mindset, trying not to overburden the development team. Like other Agile techniques, it was designed to facilitate communication and division of work inside the team, while giving live updates so each member can keep up with the process. The simple and intuitive structure helps the team work in a more efficient way.

3.1.1 Principles

Kanban is build by respecting the following 3 principles:

1. **Visualization**: Kanban uses mechanisms organised in a Kanban board. The board, by presenting all the tasks at once, in an easy to grasp manner, gives the developer a better understanding of the context, thus facilitating the workflow.
2. **Limited amount of work**: because the number of tasks are limited from a week to another (as well as the number of cards stored in production) the team does not feel overburdened with work, and is not pressured to deliver something by sacrificing on the quality aspect.
3. **Continuous flow**: when a member is done with his/her tasks, he/she can just move forward with the work process by picking up the next tasks (placed on the top of the stack). This way nobody sits around waiting to be designated with work, thus ensuring a continuous workflow.

3.1.2 Benefits

Taking in consideration the above mentioned information, it is easy to identify the way this project management method can improve the delivery process of a product, following are a couple of its advantages:

1. *Kanban methodology utilizes short working cycles.* In the case of this project the cycle was limited at one week (with an exception during the winter break). This 1 week cycles ensured a continuous delivery process, where the industry partners had the opportunity to keep the team in check, by introducing new features/corrections in the process.
2. *Kanban is very responsive to change and feedback.* As mentioned above, thanks to a short cycle, the BMW representatives had the opportunity to introduce change in an

utile time, by coming up with new features or scenarios, as well as feedback to improve or correct the direction of the project.

3. *Kanban removes time wasting activities.* While popular management techniques require a series of mandatory daily meetings and discussions, Kanban eliminates this type of time wasting activities. Meetings were organized once a week or on team's request, in cases of confusion or identification of impediments of the working process that should be discussed and solved at team level.

3.1.3 Roles

Given the Kanban approach, it does not have well predefined roles. It suggests to start from the structure of your team, and current management, and to evolve it as seen necessary, taken in consideration the specific of the methodology. As the team had no previous experience working together, it decided to have a Project Manager that would facilitate the communication between them and the supervisors, as well as take care of organizational issues. The project manager's responsibilities are:

- Managing the Kanban board (in Trello) which means writing down the tasks, assigning responsible people, adding labels, deadlines (if necessary), updating the progress, and cleaning up the board by periodically archiving the old cards.
- Managing other means of communication, like Slack, by organizing specific channels, setting up notifications and reminders.
- Setting up and managing meetings, making sure they are productive and that the team is up to date with project's progress.
- Being the spokesperson for the team, mediating the communication with the supervisors.

3.1.4 Kanban board

As already mentioned above, Kanban uses a board as it's main management tool. A Kanban board is designed to help visualise the workflow. A typical board can be broken down into the following elements:

1. Cards or Visual Signals
2. Columns
3. Work in Progress (WIP) Limits

The visual signals are usually the tasks that are written on stickies, in the case of a physical board, or on cards, in the case of software based boards. Each card encapsulates one user story, and all the user stories should require an equal or at least close enough effort. Once written down and attached to the board, these visual signals help the team, as well as other stakeholders, understand the product development. Columns represent different production stages that an user story can be in. All these stages together compose the workflow. The columns can be as simple as "To do", "In progress" and "Done", to more complicated stages, depending on the complexity of the development process. Work in Progress Limits are the maximum number of cards that can be placed in a certain column at a certain period of time. This is done in order to limit the team's ability to commit to too much work in a short period of time.

Trello as a Kanban board

For the project, the team decided, following the supervisors' suggestion, to use a digital board. This allows it not to share a physical space (like an office), and to work remotely and asynchronously. Trello is one of the simplest and easiest way to mimic a Kanban board by allowing the users to create lists, that mimic the columns, and cards that represent user stories.

For this project, it was decided to create the following workflow structure:

- Backlog - aggregates all user stories
- Breakdown (Doing and Done) - splits up user stories in smaller units of work
- Implementation (Doing and Done) - indicates what cards are currently in work
- Validation (Doing and Done) - tests the finished user stories against the definition-of-done.

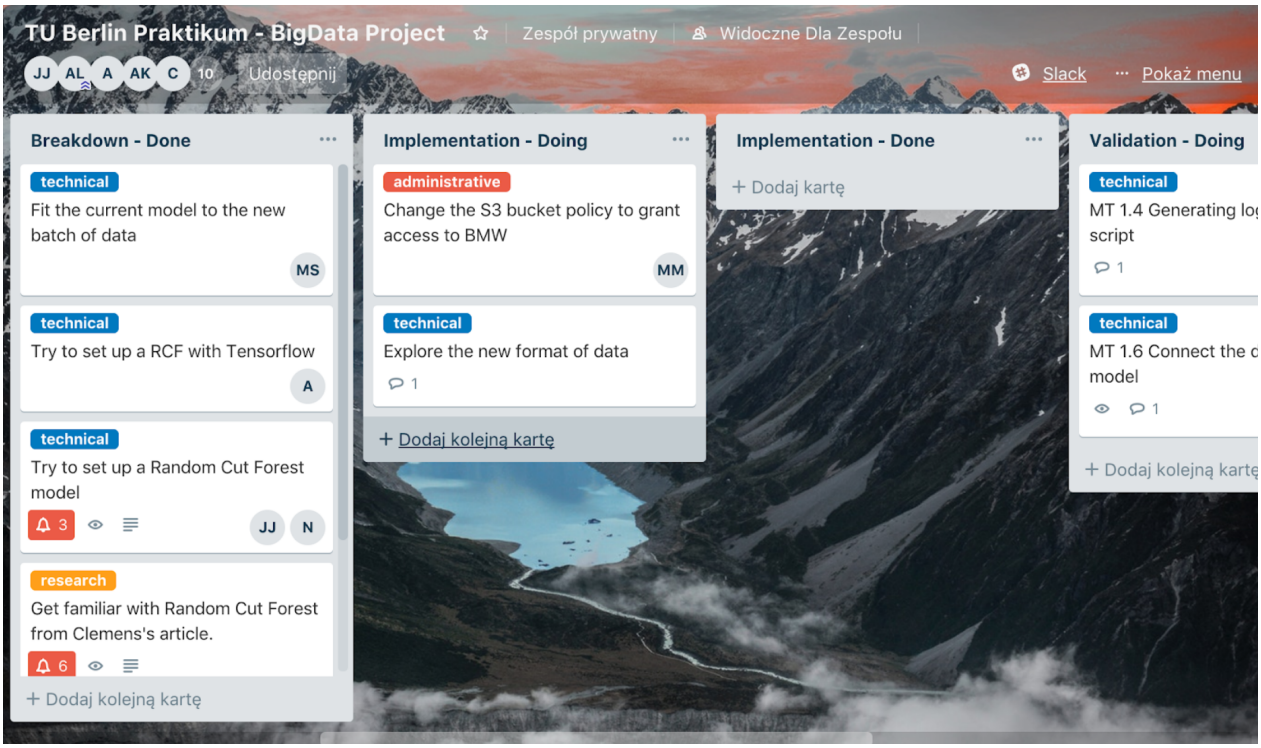


Figure 3.1: Big Data Analytics Trello board

After setting up the above mentioned columns, a user can start adding cards/user stories to the backlog. As seen in figure 3.1, Trello allows an user to add a title, a description (to detail the card), attachments, comments, due dates, responsible members and labels to a card. It's worth noting, that for this project, the team used the board not just for programming related tasks. The Trello board was used for managing all project related assignments, for the purpose of registering each week's work. In this project, the team used the following

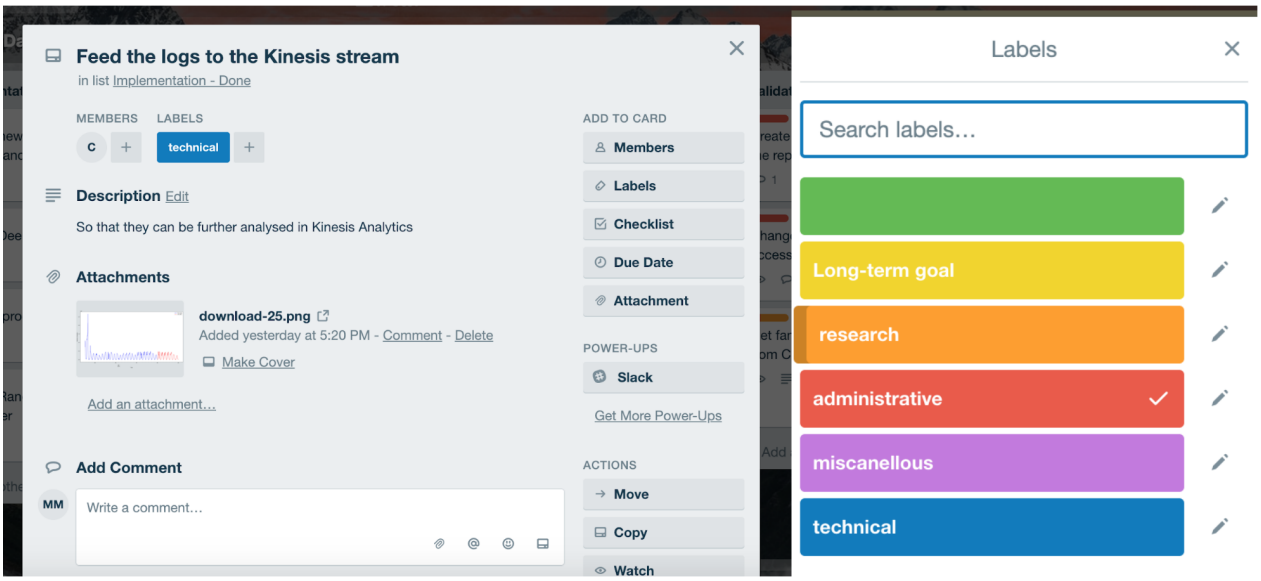


Figure 3.2: Trello board card view with labeling details

series of labels to mark the purposes of the user-story (see 3.2):

- **Long-term goal**: indicates that the card is most likely a use case in the project, which should be broken down further into smaller tasks.

- **Research**: marks tasks that require the team to look for papers, documentation, solutions to solve certain impediments.

- **Administrative**: highlights that a task has a managerial character.

- **Technical**: indicates a developer task.

- **Miscellaneous**: used for any other subjects that a task can have, except the ones mentioned above (e.g. creating the presentation or documentation paper).

3.2 Meetings

3.2.1 Internal meetings

The team organized internal weekly meetings to discuss features, plan and assign the tasks for the following production cycle. The meeting was limited at a 2 hour mark in order to keep it productive and effective and remove time wasteful discussions. The time frame in which the meeting was set, was agreed by all team members based on each individual schedule, keeping in mind that is should offer enough working horizon before the next meeting with the supervisors. Internal meetings were structured in the following way:

1. Discussing feedback from the last meeting with the supervisors
2. Adjusting the Kanban board with new tasks, or modifying the existing tasks with new requirements or details
3. Defining the goal of the following sprint/week
4. Picking up new tasks, discussing the details of each tasks, requirements and impediments
5. Assigning tasks to each team member, depending on the complexity and each member's experience with a certain technology, sometimes by forming smaller sub-teams
6. Aggregating a list of questions for the next meeting with supervisors
7. Summarizing the meeting in a 2-3 minutes recap version.

After the meeting the project manager usually sends a written recap of the session to a common Slack channel (only used by the developer team), and starts preparing an outline of the progress for the meeting with the supervisors.

3.2.2 Meetings with supervisors

To keep the process reactive and to assure that the product is in a continuous development, meetings were organized with supervisors from the TUB, BMW and AWS. The meetings followed a cyclic pattern, being organized once a week and lasting from 30 to 60 minutes (based on each party's availability). The structure of the meeting was also repetitive, building in the following way:

1. *Progress update.* The development team starts each meeting with presenting the current progress of the project, usually by demo-ing some part of the code, other times by explaining the research and investigations done in the previous week.
2. *Q and A session.* After demo, the development teams present the impediments they identified during the working process, pointing out to some issues or asking for help or assistance is solving them. Also, any uncertainties are discussed and clarified.
3. *Feedback round.* The industrial partners evaluate the progress by commenting on the current state of the project, pointing to things that can be improved. They discuss potential new features, or directions the project can go.
4. *Planning round.* Based on the received feedback, the development team comes with a proposition on things/tasks that can be achieved in the following cycle.

Chapter 4

Conclusions and Recommendations

Taking in consideration all the approaches mentioned throughout the paper for both prediction and anomaly detection models, and the fact that for BMW this project is in an early development stage, meaning that it does not possess a great amount of data to train and validate models, here are the project recommendations:

1. Start the framework by implementing a *Mean Predictor* algorithm for both anomaly detection and prediction. It trains well for small amount of data, takes less computational power, and will offer reliable results.
2. Gather data for at least a year and save all the data in different granularity buckets. Plot and test the data and see what granularity offers best results compared to the computational power needed to process it. In terms of this project, 5 minutes granularity buckets proved to perform good, preserving all the anomalies while requiring less computational power to pre-process. But the case might be different once more data is acquired.
3. Add a *geolocation* component to the data and create different data pools for each regions of the globe, instead of smashing all information into one bucket. Such a change might give new insights into the gathered information. For example, it might reveal different

holiday patterns for different countries, as well as take into consideration the specifics of holidays for different regions (christian celebration vs. Muslim celebrations vs. Asian holidays).

4. After having enough data, eventually even data that includes a location component too, run it through a *DeepAR* model. As this algorithm allows the analysis of a couple of sources of data at the same time, try including other sources of data like weather or gas prices. This algorithm (for a significant amount of data) can find some dependencies between different sources of data and significantly improve the predictions.

5. In a year or two, implement a *Random Cut Forest* model. After gathering some expert knowledge in handling anomalies, as well as information about the cause behind the anomalies and which deviations are worth looking into, you will have a better intuition about the right threshold. It might be an interesting idea to set a couple of threshold for different levels of anomalies, if this actually proves to bring any advantage in the way the anomalies are managed down the line.

6. Run the data (after at least a year) through a *Holt-Winters* model, specifically through the decomposition filter and try to identify seasonal trends (if they actually exist). Verify if the trends depend of the season/period of the year, some holidays or the geographical region. Such a model should offer new interesting insight (beyond daily and weekly patterns) into your data as well as reliable prediction results.

This being said, it's safe to say that even if new models, beyond the ones researched in this paper, are implemented, the general pipeline still stands valid and is flexible enough to be customized to the needs of BMW.

Appendices

Appendix A

Real-time Anomaly Detection in VPC Flow Logs (in AWS)

A.1 Introduction

Credit goes to Igor Kantor (https://medium.com/@devfire) who wrote the original post (5 parts) on Medium:

The goal of this GitHubGist is to support anyone who wants to implement the described architecture and get it running on AWS. This means you should use both the Medium Post and this GitHubGist for the implementation (since I will not repeat all the text here).

On my aws account I used a prefix (medium) for all services, to easily find them amongst all the other running services/instance/funtions/roles etc. (just as a suggestion). It will make cleaning up your aws account easier later on.

In this GitHubGist we will focus on the anomaly detection ("components within the red dashes"):

A.2 Part 1

https://medium.com/@devfire/real-time-anomaly-detection-in-vpc-flow-logs-part-1-introduction-55ed000e039b

Just introduction text. Does not contain any code or implementation.

A.3 Part 2

https://medium.com/@devfire/real-time-anomaly-detection-in-vpc-flow-logs-part-2-proposed-architecture-32683755abf7

Gives an overview of the architecture.

In this GitHubGist we will focus on the anomaly detection ("components within the red dashes").

A.4 Part 3

https://medium.com/@devfire/real-time-anomaly-detection-in-vpc-flow-logs-part-3-kinesis-stream-1bdd8a9426f1

This is where the fun begins.

Make sure you have the AWS Command Line Interface installed (https://docs.aws.amazon.com/de_de/cli/latest/userguide/cli-install-macos.html).

If you just installed the aws cli, make sure you run

```
aws configure
```

to connect it to your aws account.

Let's get started (with the article):

```
aws kinesis create-stream --stream-name "VPCFlowLogs" --shard-count 1

vim allowCloudWatchAccesstoKinesis.json
```

Paste the content and substitute your region as needed (in line: "Principal": "Service": "logs.**us-east-1**.amazonaws.com" ,).

Run the following command **from the same directory** where **allowCloudWatchAccesstoKinesis.json** is located:

```
aws iam create-role --role-name CloudWatchToKinesisRole --assume-role-
    policy-document file://./allowCloudWatchAccesstoKinesis.json

vim cloudWatchPermissions.json
```

Paste the content and **substitute the two resource arn strings** with yours.

Run the following command **from the same directory** where **cloudWatchPermissions.json** is located:

```
aws iam put-role-policy --role-name CloudWatchToKinesisRole --policy-
    name Permissions-Policy-For-CWL --policy-document file://./
    cloudWatchPermissions.json
```

```
1 aws logs put-subscription-filter \
2     --log-group-name "VPCFlowLogs" \
```

```
3       --filter-name "VPCFlowLogsAllFilter" \
4       --filter-pattern "[version, account_id, interface_id, srcaddr != "
            -", dstaddr != "-", srcport != "-", dstport != "-", protocol,
            packets, bytes, start, end, action, log_status]" \
5       --destination-arn "arn:aws:kinesis:us-east-1:31415926:stream/
            VPCFlowLogs" \
6       --role-arn "arn:aws:iam::31415926:role/CloudWatchToKinesisRole"

    brew install jq

    aws kinesis get-records --limit 10 --shard-iterator $(aws kinesis get-
        shard-iterator --stream-name medium_VPCFlowLogs --shard-id shardId
        -000000000000 --shard-iterator-type TRIM_HORIZON | jq -r ."
        ShardIterator") | jq -r .Records[].Data | base64 -d | zcat
```

In my case

```
aws kinesis get-records --limit 10 --shard-iterator $(aws kinesis get-
    shard-iterator --stream-name medium_VPCFlowLogs --shard-id shardId
    -000000000000 --shard-iterator-type TRIM_HORIZON | jq -r ."
    ShardIterator")
```

gives me the following:


```
{
    "Records": [],
    "NextShardIterator": "AAAAAAAAAAF3vrrgukNNTm4aHO8iRO5DXESr+
        ofQJFH57eXltrr1DTJW1DExKKckOxU5ZT2nFoQ2FOJsmfscXDRl9po0Q9Jb1Zvs9aytKcMLldmE6P7o
        /HZSC5uhlpT3/tFVAAMQAvqZ41MIFJqFik/
        kShb9q9oEB3WbYTrigihimcr1haixhIdQv2J6TJhJ6dZ1l6ggsVcnjbok1NZzIyzeABkS8fhLKSsJ/
        qIV+WmNigYX0MSYVA==",
    "MillisBehindLatest": 31385000
}
```


so when appending the

```
| jq -r .Records[].Data | base64 -d | zcat
```

part to that command it gives me "no matches found: .Records[].Data". So I still have to figure out how to insert records here (probably just need to put traffic on the vpc).

A.5 Part 4

https://medium.com/@devfire/real-time-anomaly-detection-in-vpc-flow-logs-part-4-kinesis-analytics-c80cc4977e97

Navigate to the Kinesis Analytics page in the AWS console and click on Create Application.

Name it VPCFlowLogsAnalytics.

Hook it up to the VPCFlowLogs Kinesis stream you created earlier.

in case you can not create the application because you don't have enough data see section ***Amazon Kinesis Data Generator*** *on the bottom of this file.*

Enable Lambda pre-processing and say you want a new Lambda function.

Use blueprint **kinesis-analytics-process-compressed-record** for the lambda function

Name the lambda function *KinesisAnalyticsProcessCompressedRecord*

Create in IAM a **lambda_kinesis_exec_role** and give it AmazonKinesisFullAccess.

Assign **lambda_kinesis_exec_role** for your lambda function (*KinesisAnalyticsProcessCompressedRecord*).

Increase the Timeout setting to at least 1 minute

Click on Discover Schema

In my case the schema was not recognized and I had to manually create all the columns:

I had to rename the columns **start** and **end** (see error message in screenshot). Here are the column names:

```
version, account_id, interface_id, srcaddr, dstaddr, srcport, dstport,
    protocol, packets, bytes, start_, end_, action, log_status
```

I recommend to create them in reverse order (log_status first, version last) so you dont have to sort manually.

A.6 Part 5

https://medium.com/@devfire/real-time-anomaly-detection-in-vpc-flow-logs-part-4-kinesis-analytics-c80cc4977e97

We are still where we left in part 4 (create Kinesis Analytics Application):

Click on the blue **Go to SQL Editor** button.

```
1  -- \textbf{ Anomaly detection \textbf{
2  -- Compute an anomaly score for each record in the source stream using
       Random Cut Forest
3  -- Creates a temporary stream and defines a schema
4  CREATE OR REPLACE STREAM "TEMP_STREAM" (
5  --   "APPROXIMATE_ARRIVAL_TIME"     timestamp,
6  --   "srcaddr"     varchar(16),
7  --   "dstaddr"   varchar(16),
8     "bytes"        DOUBLE,
9     "ANOMALY_SCORE"  DOUBLE,
10    "ANOMALY_EXPLANATION"  varchar(512));
11
12 -- Creates an output stream and defines a schema
13 CREATE OR REPLACE STREAM "DESTINATION_SQL_STREAM" (
14 --   "APPROXIMATE_ARRIVAL_TIME"     timestamp,
15  --  "srcaddr"     varchar(16),
16 --   "dstaddr"   varchar(16),
17    "bytes"        DOUBLE,
18    "ANOMALY_SCORE"  DOUBLE,
19    "ANOMALY_EXPLANATION"  varchar(512));
20
21 -- Compute an anomaly score for each record in the source stream
22 -- using Random Cut Forest
23 CREATE OR REPLACE PUMP "STREAM_PUMP" AS INSERT INTO "TEMP_STREAM"
```

```
24  --  SELECT STREAM "APPROXIMATE_ARRIVAL_TIME", "srcaddr", "dstaddr", "
        bytes", "ANOMALY_SCORE", "ANOMALY_EXPLANATION"
25  -- SELECT STREAM "srcaddr", "dstaddr", "bytes", "ANOMALY_SCORE", "
        ANOMALY_EXPLANATION"
26    SELECT STREAM "bytes", "ANOMALY_SCORE", "ANOMALY_EXPLANATION"
27    FROM TABLE(RANDOM_CUT_FOREST_WITH_EXPLANATION(
28      CURSOR(SELECT STREAM "bytes" FROM "SOURCE_SQL_STREAM_001"), 100,
            256, 100000, 1, true
29    )
30  );
31  -- Sort records by descending anomaly score, insert into output stream
32  CREATE OR REPLACE PUMP "OUTPUT_PUMP" AS INSERT INTO "
        DESTINATION_SQL_STREAM"
33  SELECT STREAM * FROM "TEMP_STREAM"
34  --WHERE ANOMALY_SCORE > 3.0
35  ORDER BY FLOOR("TEMP_STREAM".ROWTIME TO SECOND), ANOMALY_SCORE DESC;
```

(It is mentioned that credits go to Curtis Mitchell: https://medium.com/@curtis_mitchell)

To understand where the anomaly detection happens see:

https://docs.aws.amazon.com/kinesisanalytics/latest/sqlref/sqlrf-random-cut-forest-with-explanation.html

The following part is not from the Medium article, but was added for clarification

A.7 Part 5 continued

Create new lambda function using the blueprint **kinesis-analytics-output**

Name the lambda function **kinesis_analytics_destination**

Open your *VPCFlowLogsAnalytics* in *Kinesis Analytics applications*

Click **Connect new destination** button

Choose *AWS Lambda function* as destination and select **kinesis_analytics_destination**

As *In-application stream name* choose **DESTINATION_SQL_STREAM**

Click **Save and continue**

A.8 Amazon Kinesis Data Generator (used for part 4)

To supply the Kinesis Stream with data I had to set up an **Amazon Kinesis Data Generator** first.

Open the page https://awslabs.github.io/amazon-kinesis-data-generator/web/help.html.

Download the CloudFormation template https://s3-us-west-2.amazonaws.com/kinesis-helpers/cognito-setup.json

Click the blue **Create a Cognito User with CloudFormation** button (on the https://awslabs.github.io/amazon-kinesis-data-generator/web/help.html).

You are redirected to aws console.

Choose **Upload a template to Amazon S3** and choose the **cognito-setup.json** which you downloaded before.

Click the **next** button (bottom right).

Read the rest of the https://awslabs.github.io/amazon-kinesis-data-generator/web/help.html to learn how to access and use the Amazon Kinesis Data Generator.

A.9 allowCloudWatchAccesstoKinesis.json

```
1  {
2    "Statement": {
3      "Effect": "Allow",
4      "Principal": { "Service": "logs.us-east-1.amazonaws.com" },
5      "Action": "sts:AssumeRole"
6    }
7  }
```

A.10 cloudWatchPermissions.json

```
1  {
2    "Statement": [
3      {
4        "Effect": "Allow",
5        "Action": "kinesis:PutRecord",
6        "Resource": "arn:aws:kinesis:us-east-1:31415926:stream/
             VPCFlowLogs"
7      },
8      {
9        "Effect": "Allow",
10       "Action": "iam:PassRole",
11       "Resource": "arn:aws:iam::31415926:role/CloudWatchToKinesisRole"
12     }
13   ]
14 }
```

A.11 Authors per section

Section number	Section title	authors
1	Introduction	Marina
2.0.1	Introduction	Jacek 30% , Alexandre 70%
2.0.2	High-level description of the pipeline architecture	Jacek 70% , Alexandre 30%
2.1.1	Introduction	Alexandre
2.1.2	Data Generation	Ali, Nursultan
2.1.3	Fixed data approach	Marius
2.1.4	Data streaming approach	Alexandre
2.2	Introduction	Marius
2.2.1	Amazon SageMaker Random Cut Forest	Nursultan
2.2.2	Mean Predictor Algorithm	Marius
2.2.3	Real-time anomaly detection with Kinesis	Clemens
2.3	Introduction	Marius
2.3.1	Mean Predictor Model	Marius
2.3.2	DeepAR Model	Alexandre
2.3.3	Holt-Winter's Method	Marina
2.4	Data Management	Ali, Jacek
2.5	Notification functions	Jacek
2.6	Project delivery	Alexandre
3	Project Management Techniques	Marina
4	Conclusions and recommendations	Marina

Table A.1: Authors by section

Bibliography

[1] *Amazon Simple Notification Service*. URL: https://aws.amazon.com/sns/.

[2] *AWS Kinesis Data Streams*. URL: https://aws.amazon.com/kinesis/data-streams/.

[3] *AWS Lambda*. URL: https://aws.amazon.com/lambda/.

[4] *AWS - Random Cut Forest with explanation*. URL: https://docs.aws.amazon.com/kinesisanalytics/latest/sqlref/sqlrf-random-cut-forest-with-explanation.html.

[5] *AWS Sagemaker Random Cut Forest*. URL: https://docs.aws.amazon.com/sagemaker/latest/dg/randomcutforest.html.

[6] *DeepAR Forecasting algorithm*. URL: https://docs.aws.amazon.com/sagemaker/latest/dg/deepar.html.

[7] Charles C. Holt. "Forecasting seasonals and trends by exponentially weighted moving averages". In: *International Journal of Forecasting* 20.1 (2004), pp. 5–10. DOI: https://doi.org/10.1016/j.ijforecast.2003.09.015.

[8] *Netflix Quote*. URL: https://medium.com/@devfire/real-time-anomaly-detection-in-vpc-flow-logs-part-2-proposed-architecture-32683755abf7.

[9] *Real-time Anomaly Detection in VPC Flow Logs*. URL: https://medium.com/@devfire/real-time-anomaly-detection-in-vpc-flow-logs-part-1-introduction-55ed000e039b.

[10] *Subscription Filters with Amazon Kinesis Data Firehose*. URL: https://docs.aws.amazon.com/AmazonCloudWatch/latest/logs/SubscriptionFilters.html#FirehoseExample.

[11] *What is Amazon SageMaker ?* URL: https://docs.aws.amazon.com/sagemaker/latest/dg/whatis.html.